Programação orientada a objetos

2ª Edição
Abrange Python 2.3

Aprendendo

Python

O'REILLY®

Mark Lutz & David Ascher

LINGUAGEM DE PROGRAMAÇÃO

ARNOLD, GOSLING & HOLMES
A Linguagem de Programação Java, 4.ed.

DEITEL & DEITEL
C++, Como Programar, 3.ed.

DEITEL, DEITEL, NIETO, LIN & SADHU
XML, Como Programar

DEITEL, DEITEL, NIETO & MCPHIE
Perl, Como Programar

FLANAGAN, D.
Java: O Guia Essencial, 5.ed.

FLANAGAN, D.
JavaScript: O Guia Definitivo, 4.ed

GALUPPO, SANTOS & MATHEUS
Desenvolvendo com C#

***HERRINGTON, J. D.**
Dicas de PHP

HORSTMANN, C.
Big Java

HORSTMANN, C.
Conceitos de Computação com o Essencial de C++, 3. ed

HORSTMANN, C.
Conceitos de Computação com o Essencial de Java, 3. ed

HUBBARD, J.
Programação em C++, 2.ed.
(COLEÇÃO SCHAUM)

HUBBARD, J.
Programação com Java, 2.ed.
(COLEÇÃO SCHAUM)

LIPPMAN, S.
C#: Um Guia Prático

LUTZ & ASCHER
Aprendendo Python, 2.ed.

***ROBBINS & BEEBE**
Classic Shell Scripting

ROMAN, AMBLER & JEWELL
Dominando o Enterprise JavaBeans, 2.ed.

SIEVER & COLS
Linux: O Guia Essencial

STROUSTRUP, B.
A Linguagem de Programação C++, 3.ed.

TITTEL, E.
XML (COLEÇÃO SCHAUM)

** Livros em produção no momento da impressão desta obra, mas que muito em breve estarão à disposição dos leitores em língua portuguesa.*

Bookman® Editora
Av. Jerônimo de Ornelas, 670
90040-340 Porto Alegre, RS, Brasil
Fone (51) 3027-7000 Fax (51) 3027-7070
e-mail: bookman@artmed.com.br

O'REILLY®

Recorte e use como um marcador de página

Aprendendo

Python

Mark Lutz é instrutor de Python, escritor e desenvolvedor de software independente. É autor dos livros *Programming Python* e *Python Pocket Reference*, da O'Reilly. Mark trabalha com Python desde 1992. Começou a dar aulas em 1997 e foi instrutor em mais de 90 sessões de treinamento sobre Python, até o início de 2003. Além disso, tem bacharelado e mestrado em ciência da computação pela Universidade de Winconsin e nas duas últimas décadas trabalhou com compiladores, ferramentas de programação, aplicações de script e diversos sistemas cliente/servidor. Quando Mark faz uma pausa na divulgação do Python, leva uma vida normal com seus filhos no Colorado, EUA. Mark pode ser encontrado por email, no endereço *lutz@rmi.net* ou na Web, no endereço *http://www.rmi.net/~lutz*.

David Ascher é pesquisador, mas passou os últimos anos envolvido com o desenvolvimento de software, preocupado com o fornecimento de ferramentas profissionais para programadores que usam linguagens de programação de código-fonte aberto. Após liderar a criação do ambiente de desenvolvimento integrado Komodo (para programadores de Python, dentre outros, e escrito principalmente em Python), agora ele é responsável pela direção geral da divisão ActiveState da Sophos, que produz distribuições de linguagem, ferramentas de desenvolvimento e serviços para Python, Perl, Tcl, PHP e outras linguagens. David é também diretor da Python Software Foundation desde seu início e ajuda a organizar conferências e outros eventos sobre Python.

L975a Lutz, Mark
Aprendendo python / Mark Lutz, David Ascher ; tradução João Tortello. – 2. ed. – Porto Alegre : Bookman, 2007
568 p. ; 25 cm.

ISBN 978-85-7780-013-1

1. Informática – Linguagem de Programação - Python. I. Ascher, David. II. Título.

CDU 004.43PYTHON

Catalogação na publicação: Juliana Lagôas Coelho – CRB 10/1798

Mark Lutz & David Ascher

Aprendendo

Python

Tradução:

João Tortello

Consultoria, supervisão e revisão técnica desta edição:

Rodrigo Camarão

Profissional com certificações MCSE – MCT– MCITP– CLP

Instrutor da Sisnema Informática

Reimpressão 2008

2007

Ao falecido Frank Willison, nosso mentor,
amigo e primeiro editor.

Prefácio

Este livro é uma introdução à linguagem de programação Python. O Python é uma linguagem orientada a objetos utilizada em uma variedade de domínios, tanto para programas independentes como para aplicações de script. Ela é gratuita, portável, poderosa e fácil de usar.

Seja você iniciante na programação ou desenvolvedor profissional, o objetivo deste livro é ensiná-lo o básico do Python rapidamente.

A SEGUNDA EDIÇÃO

Nos quatro anos após a publicação da primeira edição deste livro, no final de 1998, houve alterações significativas, tanto na linguagem Python quanto nos assuntos apresentados pelos autores nas sessões de treinamento em Python. Embora tenhamos tentado manter o máximo possível da versão original, esta nova edição reflete as alterações recentes feitas no Python e no treinamento.

Com relação à linguagem, esta edição foi completamente atualizada para refletir o Python 2.2 e todas as suas mudanças. Além disso, foi incorporada uma discussão sobre alterações antecipadas a serem feitas na futura versão 2.3. Alguns dos principais tópicos da linguagem, para os quais você encontrará abordagem nova ou ampliada nesta edição, são:

- Compreensão de lista (Capítulo 14)
- Exceções de classe (Capítulo 25)
- Métodos de string (Capítulo 5)
- Atribuição ampliada (Capítulo 8)
- Divisão clássica, real e de base (Capítulo 4)
- Importações de pacote (Capítulo 17)
- Escopos de função aninhados (Capítulo 13)
- Funções geradoras e iteradores (Capítulo 14)

- Strings em Unicode (Capítulo 5)
- Tipos de subclasse (capítulos 7 e 23)
- Métodos estáticos e de classe (Capítulo 23)
- Atributos de classe pseudo-privados (Capítulo 23)
- Instruções print e import estendidas (capítulos 8 e 18)
- Novas funções internas, como `zip` e `isinstance` (capítulos 7 e 10)
- Classes de estilo novo (Capítulo 23)\
- Novas configurações, opções de execução e arquivos *.pth* (Capítulo 3 e Apêndice A)
- Novas ferramentas de configuração, como IDLE, Psyco, Py2exe e Installer (Capítulos 2, 3 e 29)

Pequenas alterações feitas na linguagem (por exemplo, promoção de inteiro longo, listas de exportação de módulo) aparecem por todo o livro. Além dessas alterações na linguagem, ampliamos as partes referentes à linguagem básica desta edição (partes I a VII), com novos tópicos e exemplos apresentados nas sessões de treinamento em Python conduzidas por Mark nos últimos anos. Por exemplo, você encontrará:

- Uma nova introdução à POO (Capítulo 19)
- Um novo panorama sobre tipagem dinâmica (Capítulo 4)
- Um novo resumo das ferramentas de desenvolvimento (Capítulo 26)
- Material novo sobre arquitetura e execução de programas (capítulos 2, 3 e 15)
- Nova abordagem das fontes de documentação (Capítulo 11)

Muitas adições e alterações foram feitas na parte referente à linguagem básica, tendo em vista os iniciantes. Você verá também que a abordagem de muitos tópicos da linguagem básica foram substancialmente ampliados nesta edição, com novas discussões e exemplos. Como este texto se tornou o principal recurso para aprender a linguagem Python básica, tomamos a liberdade de tornar essa abordagem mais completa e adicionamos novos casos de uso. Do mesmo modo, atualizamos a Parte VII para refletir os recentes domínios de aplicação do Python e padrões de utilização modernos.

Além disso, esta edição inteira integra um novo conjunto de dicas e truques do Python, extraídos de aulas lecionadas nos últimos sete anos e do uso do Python na última década. Os exercícios foram atualizados e ampliados para refletir a prática corrente no Python, novos recursos da linguagem e erros comuns dos iniciantes que testemunhamos em primeira mão nos últimos anos. De modo geral, esta edição é maior, tanto porque o Python é maior como porque adicionamos contexto que se mostrou ser importante na prática.

Um vez que esta edição é mais completa, dividimos a maior parte dos capítulos originais em trechos menores. Isto é, reorganizamos a seção da linguagem básica em muitas partes, com vários capítulos, para facilitar a abordagem. Os tipos e as instruções, por exemplo, agora são duas partes de alto nível, com um capítulo para cada tópico importante sobre tipo e instrução. Essa nova estrutura é projetada para tornar possível passar mais informações, sem intimidar os leitores. Nesse processo, os exercícios e problemas foram movidos dos finais dos capítulos para os finais das partes; agora eles aparecem no fim do último capítulo de cada parte.

Apesar dos assuntos novos, este livro ainda é voltado para os iniciantes em Python e foi concebido como um primeiro texto sobre a linguagem para os programadores.* Ele preserva grande parte do material, da estrutura e do foco da primeira edição. Onde apropriado, ampliamos as introduções para os iniciantes e isolamos os novos tópicos mais avançados da linha de assunto da discussão principal, para não obscurecer os fundamentos. Além disso, como é principalmente baseada na experiência em treinamento e em materiais comprovados, esta edição, assim como a primeira, ainda pode servir como uma aula introdutória, em que o leitor pode ditar o seu ritmo, sobre Python.

PRÉ-REQUISITOS

Não há pré-requisitos para a leitura deste livro. Ele já foi usado com sucesso por iniciantes e por veteranos de programação. Em geral, contudo, consideramos que qualquer exposição à programação ou produção de scripts anterior a este texto pode ser útil, mesmo não sendo exigido de todos os leitores.

Este livro é projetado para ser um texto de nível introdutório sobre Python. Pode não ser o texto ideal para alguém que nunca mexeu em um computador (não vamos dizer o que é um computador), mas não fizemos nenhuma suposição sobre sua experiência ou treinamento em programação.

Por outro lado, não presumimos que os leitores nada sabem; é fácil fazer coisas úteis em Python e esperamos mostrar isso a você. De vez em quando, o texto contrasta o Python com linguagens como C, C++, Java e Pascal, mas você pode ignorar essas comparações, se não tiver usado essas linguagens.

Algo que provavelmente devemos mencionar antecipadamente: o criador do Python, Guido van Rossum, escolheu o nome em homenagem à série humorística *Monty Python's Flying Circus*, da BBC. É claro que esse legado acrescentou um tempero jocoso em muitos exemplos do Python. Por exemplo, os tradicionais "foo" e "bar" se tornam "spam" e "eggs" no mundo Python e em algum código que você verá neste livro. Do mesmo modo, os ocasionais "Brian", "Ni" e "shrubbery" devem suas aparições a esse homônimo. Você não precisa conhecer a série para entender os exemplos (símbolos são símbolos).

A ABRANGÊNCIA DESTE LIVRO

Embora este livro aborde todos os fundamentos da linguagem Python, mantivemos sua abrangência limitada em favor da velocidade e do tamanho. Para manter as coisas simples, este livro focaliza os conceitos básicos, utiliza exemplos pequenos e auto-suficientes para ilustrar os pontos e, às vezes, omite os pequenos detalhes que estão prontamente disponíveis nos manuais de referência. Por isso, este livro provavelmente é melhor descrito como uma introdução e um degrau para textos mais avançados e completos.

Por exemplo, não falamos muito sobre a integração Python/C – um assunto complexo e fundamental para muitos sistemas baseados em Python. Também não falamos muito sobre a história do Python ou seus processos de desenvolvimento. E as aplicações populares do Python, como GUIs, ferramentas de sistema e scripts de rede recebem um pequeno levantamento, se mencionados. Naturalmente, esse escopo reduzido deixa alguma coisas de fora.

* "Programador" é qualquer um que já tenha escrito uma linha de código em qualquer linguagem de programação ou de script. Mesmo que você não se enquadre nessa classificação, ainda assim pode achar que este livro é útil. Mas nós vamos passar mais tempo ensinando Python do que fundamentos da programação.

De modo geral, o Python eleva em alguns pontos a marca da qualidade no mundo dos scripts. Algumas de suas idéias exigem mais contexto do que podemos fornecer aqui e seríamos desleixados se não recomendássemos mais estudo, após você terminar este livro. Esperamos que a maioria dos leitores amplie seu entendimento de programação em nível de aplicativo a partir de outros textos.

Por causa de seu foco nos iniciantes, este livro é projetado para ser complementado por outros livros sobre Python da O'Reilly. Por exemplo, o livro *Programming Python*, segunda edição, fornece exemplos em nível de aplicativo maiores e mais avançados e foi concebido para ser um texto subseqüente para o que você está lendo agora. A segunda edição deste livro e do *Programming Python* refletem, no geral, as duas metades dos materiais de treinamento de Mark – a linguagem básica e programação de aplicativos. Além disso, o livro *Python Pocket Reference*, segunda edição, da O'Reilly, serve como um complemento de referência rápida para pesquisar os detalhes refinados que, de modo geral, pulamos aqui.

Outros livros complementares sobre Python também podem ajudar a fornecer mais exemplos, a explorar domínios específicos da linguagem e servir como referências. Recomendamos como referências os livros *Python in a Nutshell* da O'Reilly e *Python Essential Reference* da New Riders e o livro *Python Cookbook*. Independente dos livros que você escolher, deve ter em mente que o resto da história do Python exige estudar exemplos mais realistas do que aqueles para os quais há espaço aqui. Existem aproximadamente 40 livros sobre Python no idioma inglês, junto com dezenas de textos em outros idiomas. Como os livros são uma experiência subjetiva, o convidamos a examinar todos os textos disponíveis para encontrar um que esteja de acordo com suas necessidades.

Mas, apesar de sua abrangência limitada (e talvez por causa disso), achamos que você vai considerar este livro um bom primeiro texto sobre Python. Você vai aprender tudo que precisa para começar a escrever programas e scripts independentes e úteis. Quando você terminar de ler este livro, terá aprendido não apenas a linguagem em si, mas também a aplicá-la em tarefas diárias. E você estará equipado para atacar assuntos e exemplos mais avançados, quando eles aparecerem na sua frente.

ESTILO E ESTRUTURA DESTE LIVRO

Grande parte deste livro é baseada em materiais de treinamento desenvolvidos para um curso prático de três dias sobre Python. Você encontrará exercícios no final do último capítulo das partes referentes à linguagem básica, com soluções para todos eles no Apêndice B. Os exercícios são projetados para que você comece a desenvolver imediatamente e, normalmente, são um dos destaques do curso.

Recomendamos que você faça os exercícios em meio à leitura, não apenas para ganhar experiência com programação em Python, mas também porque alguns deles levantam questões não abordadas em nenhuma outra parte do livro. As soluções do Apêndice B devem ajudar, caso você fique travado (e o encorajamos a "colar" o quanto quiser e quantas vezes for necessário). Naturalmente, você precisará instalar o Python para executar os exercícios.

Como este texto é projetado para introduzir os fundamentos da linguagem rapidamente, organizamos a apresentação por recursos importantes da linguagem e não por exemplos. Adotamos uma estratégia de baixo para cima aqui: dos tipos de objeto internos para instruções, unidades de programa e assim por diante. Cada capítulo é totalmente independente, mas os

capítulos posteriores usam idéias apresentadas nos anteriores (por exemplo, quando chegamos à classes, presumimos que você sabe como escrever funções); portanto, uma leitura linear faz mais sentido. A partir de uma perspectiva mais ampla, este livro está dividido nas seguintes áreas funcionais e suas partes correspondentes.

Linguagem básica

Esta parte do livro apresenta a linguagem Python usando uma estratégia de baixo para cima. Ela é organizada com uma parte por recurso importante da linguagem – tipos, funções e assim por diante – e, em sua maior parte, os exemplos são pequenos e auto-suficientes (alguns também poderiam chamar os exemplos desta seção de artificiais, mas eles ilustram os pontos que queremos explicar). Esta seção representa o grosso do texto, o que lhe diz algo sobre o enfoque do livro. Ela é composta das seguintes partes:

Parte I, *Começando*

Começamos com uma visão geral do Python, que responde às perguntas iniciais – por que as pessoas usam a linguagem, para que ela serve etc. O primeiro capítulo apresenta as principais idéias subjacentes à tecnologia para garantir algum contexto básico.

Em seguida, tem início a parte técnica do livro, explorando as maneiras como você e o Python executam programas. Seu objetivo é fornecer informações suficientes para trabalhar nos exemplos e exercícios posteriores. Se você precisar de mais ajuda, detalhes adicionais da configuração estão disponíveis no Apêndice A.

Parte II, *Tipos e operações*

Nosso passeio pela linguagem começa com um estudo aprofundado de seus principais tipos de objeto internos: números, listas, dicionários etc. Você pode fazer muita coisa no Python apenas com essas ferramentas.

Parte III, *Instruções e sintaxe*

A parte seguinte apresenta as instruções do Python – o código que você digita para criar e processar objetos. Ela também apresenta o modelo de sintaxe geral da linguagem.

Parte IV, *Funções*

Aqui tem início nosso estudo das ferramentas de estrutura de programa de nível mais alto do Python. As funções mostram ser uma maneira simples de empacotar código para reutilização.

Parte V, *Módulos*

Os módulos do Python permitem que você organize instruções e funções em componentes maiores e esta parte ilustra como criar, usar e recarregar módulos.

Parte VI, *Classes e POO*

Aqui, exploramos a ferramenta de programação orientada a objetos (POO) do Python, a classe. Conforme você verá, no Python, a POO está relacionada principalmente com pesquisa de nomes em objetos vinculados.

Parte VII, *Exceções e ferramentas*

Encerramos a cobertura da linguagem básica desta seção do livro com um exame do modelo e instruções de tratamento de exceção do Python, e um breve panorama das ferramentas de desenvolvimento. Isso aparece por último porque as exceções podem ser classes, se você quiser.

Camadas externas

A Parte VIII mostra as ferramentas internas do Python e as utiliza em um conjunto de pequenos exemplos de programa. As tarefas comuns são demonstradas no Python para fornecer a você algum contexto do mundo real, usando a linguagem em si e suas bibliotecas e ferramentas padrão.

Capítulo 27, *Tarefas comuns no Python*
: Este capítulo apresenta uma seleção dos módulos e funções incluídos na instalação padrão do Python. Por definição, eles compreendem o conjunto mínimo de módulos aos quais você pode razoavelmente esperar que qualquer usuário de Python tenha acesso. Conhecer o conteúdo desse conjunto de ferramentas padrão fará com que você economize semanas de trabalho.

Capítulo 28, *Modelos*
: Este capítulo apresenta alguns aplicativos reais. Construindo a linguagem básica explicada nas partes anteriores e as ferramentas internas descritas no Capítulo 27, apresentamos muitos programas pequenos, mas úteis, que mostram como reunir tudo. Abordamos três áreas de interesse para a maioria dos usuários de Python: tarefas básicas, processamento de texto e interfaces de sistema. Fechamos com uma breve discussão sobre o Jython, a implementação Java do Python, e um programa significativo nessa linguagem.

Capítulo 29, *Recursos do Python*
: Este capítulo discute as camadas da comunidade Python e bibliotecas especializadas que fazem parte da distribuição do Python padrão ou estão gratuitamente disponíveis em outros fornecedores.

Apêndices

O livro termina com apêndices que fornecem dicas específicas para usar Python em várias plataformas (Apêndice A) e fornece soluções para os exercícios que aparecem no final do último capítulo de cada parte (Apêndice B). Note que o índice e o sumário podem ser usados para procurar detalhes, mas não existem apêndices de referência neste livro. Conforme mencionado anteriormente, o livro *Python Pocket Reference*, segunda edição (O'Reilly), assim como outros livros e os manuais de referência gratuitos do Python, mantidos no endereço *http://www.python.org*, contêm os detalhes da sintaxe e das ferramentas internas.

Atualizações do livro

Aprimoramentos acontecem (e o mesmo ocorre com erros de digi`^H^H^H`ação). Atualizações, complementos e correções deste livro serão mantidos (ou referenciados) na Web em um dos seguintes sites:

- *http://www.oreilly.com* (site da O'Reilly)
- *http://www.rmi.net/~ lutz* (site de Mark)
- *http://starship.python.net/~da* (site de David)
- *http://www.python.org* (site principal do Python)
- *http://www.rmi.net/~lutz/about-lp.html* (página Web do livro)

Se pudéssemos ser mais clarividentes, seríamos, mas a Web muda mais rápido do que os livros impressos.

CONVENÇÕES DE FONTE

Este livro usa as seguintes convenções tipográficas:

Itálico

Para endereços de email, nomes de arquivo, URLs, para enfatizar termos novos quando aparecem pela primeira vez e para alguns comentários dentro de seções de código

`Largura constante`

Mostra o conteúdo de arquivos ou a saída de comandos e serve para designar módulos, métodos, instruções e comandos

`Largura constante em negrito`

Em seções de código, para mostrar comandos ou texto que seriam digitados

`Largura constante em itálico`

Mostra substituíveis em seções de código

`<Largura constante>`

Representa unidades sintáticas que você substitui por código real

Indica uma dica, sugestão ou nota geral relacionada ao texto próximo

Indica um alerta ou aviso relacionado ao texto próximo

Em nossos exemplos, o caractere % no início de uma linha de comando de sistema significa o prompt do sistema, qualquer que possa ser em sua máquina (por exemplo, `C:\Python22>`, em uma janela DOS). Não digite o caractere %! Analogamente, em listagens de interação do interpretador, não digite os caracteres >>> e... mostrados no início das linhas – eles são prompts exibidos pelo Python. Digite apenas o texto após esses prompts. Para ajudá-lo a lembrar disso, as entradas de usuário são mostradas em negrito neste livro. Além disso, normalmente você não precisa digitar texto que começa com # nas listagens; conforme explicaremos posteriormente, esses são comentários e não código executável.

SOBRE OS PROGRAMAS DESTE LIVRO

Este livro e todos os exemplos de programa nele constantes são baseados no Python versão 2.2 e refletem a futura versão 2.3. Mas, como usaremos a linguagem básica, você pode estar certo de que a maior parte do que temos a dizer não mudará muito nas versões posteriores do Python. A maior parte deste livro também se aplica às versões anteriores do Python; naturalmente, se você tentar usar extensões adicionadas depois da versão que possui, todas as apostas estarão perdidas. Como regra geral, o Python mais recente é o melhor. Como este livro focaliza a linguagem básica, a maior parte dele também se aplica ao Jython, a implementação da linguagem Python baseada em Java, assim como a outras implementações de Python descritas no Capítulo 2.

O código-fonte dos exemplos do livro, assim como as soluções dos exercícios, podem ser buscados no seu site da Web, no endereço *http://www.oreilly.com/catalog/lpython2/*. Então, como você executa os exemplos? Vamos entrar nos detalhes da inicialização no Capítulo 3; portanto, fique ligado para os detalhes sobre isso.

USANDO EXEMPLOS DE CÓDIGO

Este livro está aqui para ajudá-lo a fazer seu trabalho. Em geral, você pode usar o código deste livro em seus programas e em sua documentação. Você não precisa entrar em contato conosco para pedir permissão, a não ser que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que usa vários trechos de código deste livro não exige permissão. Vender ou distribuir um CD-ROM de exemplos dos livros da O'Reilly *exige* permissão. Responder a uma pergunta citando este livro e copiar um exemplo de código não exige permissão. Incorporar um volume significativo de exemplo de código deste livro na documentação de seu produto *exige* permissão.

Apreciamos, mas não exigimos, que a fonte seja citada. Normalmente, uma referência inclui o título, autor, editora e ISBN. Por exemplo, "*ActionScript: The Definitive Guide*, Second Edition, by Colin Moock. Copyright 2001 O'Reilly & Associates, Inc., 0-596-00369-X".

Se você achar que seu uso dos exemplos de código fica fora do uso desimpedido ou da permissão dada acima, fique à vontade para entrar em contato conosco no endereço *permissions@oreilly.com*

COMO ENTRAR EM CONTATO CONOSCO

Envie seus comentários e perguntas a respeito deste livro para a editora:*

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (nos Estados Unidos e Canadá)
(707) 829-0515 (internacional ou local)
(707) 829-0104 (fax)

Temos uma página na Web para este livro, onde listamos errata, exemplos e informações adicionais. Você pode acessar essa página no endereço:

http://www.oreilly.com/catalog/lpython2

Para fazer comentários ou perguntas técnicas sobre este livro, envie email para:

bookquestions@oreilly.com

Para obter mais informações sobre nossos livros, conferências, Centros de Recurso e a O'Reilly Network, veja nosso site na Web, no endereço:

http://www.oreilly.com

Mark e David também ficarão contentes em responder perguntas dos leitores sobre o livro; contudo, é mais provável que você receba uma resposta enviando perguntas sobre a "Linguagem Básica" para Mark e sobre a "Camada Externa" para David, as duas respectivas áreas principais dos autores. Você pode encontrar os endereços de email dos dois autores no site da Web do livro.

(Ao logo do livro, normalmente usamos "nós" para nos referirmos aos dois autores, mas ocasionalmente introduzimos o nome de um autor específico – normalmente, Mark nas partes da Linguagem Básica e David nas partes das Camadas Externas, refletindo o principal autor de cada parte. Embora este livro tenha sido o esforço conjunto de muitos, às vezes cada autor se desvia do coletivo.)

* N. de R.T.: Comentários sobre a edição brasileira podem ser encaminhados para secretariaeditorial@artmed.com.br.

AGRADECIMENTOS

Gostaríamos de expressar nossa gratidão a todas as pessoas que tomaram parte no desenvolvimento deste livro. Primeiramente, gostaríamos de agradecer às editoras que trabalharam neste projeto: Laura Lewin, Paula Ferguson e, finalmente, Linda Mui. Também gostaríamos de agradecer à O'Reilly em geral, por apoiar outro projeto de livro sobre Python. Temos o prazer de fazer parte do que agora é uma linha de produto completa e crescente sobre Python na O'Reilly.

Agradecemos também a todos os que tomaram parte na primeira revisão deste livro – Guido van Rossum, Alex Martelli, Anna Ravenscroft, Sue Giller e Paul Prescod.

E, por criarem uma linguagem tão agradável e útil, devemos muito a Guido van Rossum e ao restante da comunidade Python; assim como a maioria dos sistemas de código-fonte aberto, o Python é o produto de muitos esforços.

Também gostaríamos de agradecer especialmente ao nosso editor original deste livro, o falecido Frank Willison. Frank teve um profundo impacto sobre o mundo do Python e sobre nossas próprias carreiras pessoais. Não é exagero dizer que Frank foi responsável por grande parte da graça e do sucesso dos primeiros dias do Python. Na verdade, este livro foi idéia dele. Em reconhecimento à sua visão e amizade, dedicamos esta nova edição a ele. Vamos trabalhar, Frank.

Mark também diz:

Quando fui apresentado ao Python pela primeira vez, em 1992, não tinha idéia do impacto que ele teria na década seguinte da minha vida. Após escrever a primeira edição deste livro, em 1995, comecei a viajar pelo país e pelo mundo, ensinando Python para iniciantes e especialistas. Desde a conclusão da primeira edição deste livro, em 1999, tenho sido instrutor e escritor independente de Python, em tempo integral, graças principalmente à popularidade exponencialmente crescente da linguagem.

Até o início de 2003, lecionei em aproximadamente 90 sessões de treinamento em Python, nos EUA, Europa, Canadá e México, e conheci mais de mil alunos. Além de acumular milhagem para vôos, essas aulas me ajudaram a refinar minhas colaborações para este livro, especialmente o material da linguagem básica. De modo geral, essas partes do livro vêm diretamente dos meus materiais de curso atuais.

Gostaria de agradecer a todos os alunos que participaram dos meus cursos nos últimos sete anos. Junto com as recentes alterações no Python, seu retorno desempenhou um papel muito importante na moldagem das minhas contribuições para este texto. Não há nada mais instrutivo do que observar mil alunos repetindo os mesmos erros de iniciantes! A seção sobre a linguagem básica desta segunda edição deve suas alterações principalmente às aulas dadas após 1999; gostaria de destacar a Hewlett-Packard, Intel e Seagate pelas várias sessões nesse período. E gostaria de agradecer especialmente aos clientes que foram os anfitriões das aulas em Dublin, Cidade do México, Barcelona e Porto Rico; seria difícil imaginar pessoais mais hábeis em servir café de máquina.

Gostaria de agradecer a O'Reilly por me dar a chance de trabalhar em, agora, seis projetos de livro; foi muito divertido (e só parece um pouco com o filme *Groundhog Day*). Também gostaria de agradecer ao co-autor David Ascher por seu trabalho e paciência neste projeto. Além deste livro e de seu trabalho cotidiano desenvolvendo ferramentas Python na ActiveState, David também dedica tempo para organizar conferências, editar outros livros e muito mais.

Finalmente, algumas notas pessoais de agradecimento. A todas as pessoas com quem trabalhei em várias empresas ao longo da minha carreira. À biblioteca pública de Boulder, na qual me escondi enquanto escrevia partes desta edição. Ao falecido Carl Sagan, pela inspiração na juventude. A Jimmy Buffet, pela perspectiva no começo da meia-idade. Para uma mulher do Novo México, em um vôo de Oklahoma, por me lembrar da importância de ter um sonho. Para o Denver Broncos, por vencer o campeonato (duas vezes). Para a Sharp e para a Sony, por fazerem aquelas máquinas fantásticas. E, acima de tudo, para meus filhos, Michael, Samantha e Roxanne, por me tornarem um homem realmente rico.

Longmont e Boulder, Colorado

David também diz:

Além dos agradecimentos anteriores, gostaria de agradecer especialmente aos seguintes.

Primeiro, agradeço a Mark Lutz por me convidar a trabalhar com ele neste livro e por apoiar meus esforços como instrutor de Python. Agradeço também ao impressionante grupo de pessoas ligadas ao Python que me estimularam a entender a linguagem no princípio, especialmente a Guido, Tim Peters, Don Beaudry e Andrew Mullhaupt. É espantoso como um pequeno estímulo no momento certo pode ter um impacto tão duradouro.

Eu ensinava Python e Jython, como Mark ainda faz. Os alunos desses cursos me ajudaram a identificar as partes do Python mais difíceis de aprender, assim como me lembraram dos aspectos da linguagem que a tornam tão agradável de usar, e agradeço a eles por seu retorno e estímulo. Também gostaria de agradecer àqueles que me deram a chance de desenvolver esses cursos: Jim Anderson (Brown University), Cliff Dutton (então na Distributed Data Systems), Geoffrey Philbrick (então na Hibbitt, Karlson & Sorensen, Inc.), Paul Dubois (Lawrence Livermore National Labs) e Ken Swisz (KLA-Tencor). Embora eu não esteja mais ensinando Python como rotina, ainda conto com essa experiência ao instruir iniciantes.

Agradeço aos meus conselheiros científicos, Jim Anderson, Leslie Welch e Norberto Grzywacz, todos os quais gentilmente apoiaram meus esforços com o Python em geral e neste livro em particular, não necessariamente entendendo por que eu estava fazendo isso, mas confiando em mim, todavia. Todas as lições que me ensinaram ainda são relevantes diariamente.

As primeiras vítimas de meus esforços de evangelização em Python merecem estrelas de ouro por tolerarem meus mais entusiásticos (alguns poderiam dizer fanáticos) primeiros dias: Thanassi Protopapas, Gary Strangman e Steven Finney. Thanassi também deu seu útil retorno em um rascunho inicial do livro. Vários Activators (conhecidos dos civis como "funcionários da ActiveState") foram tremendos colegas e amigos nestes últimos três anos – destacarei Mark Hammond, Trent Mick, Shane Caraveo e Paul Prescod. Cada um deles me ensinou muito sobre Python e muito mais. A ActiveState como um todo me proporcionou um ambiente espantoso para construir uma carreira, aprender coisas novas com pessoas fascinantes todos os dias e ainda programar em Python.

Agradeço à minha família: meus pais, JacSue e Philippe, por sempre me estimularem a fazer o que eu queria; meu irmão Ivan, por me lembrar de alguns dos meus primeiros encontros com textos de programação (após horas de esforço, perceber que uma listagem de programa na revista Byte tinha erros fazia um garoto de 13 anos chorar de frustração); minha esposa Emily, por seu apoio constante e pela crença incondicional de que escrever

um livro era algo que eu poderia fazer; nossos filhos, Hugo e Sylvia, por compartilharem os computadores comigo – eles encaram os computadores com tanta naturalidade que mal posso esperar para ver o que a geração deles fará.

Finalmente, pensando nesta edição em particular, quero agradecer a todos os que colaboraram com a comunidade Python. É extraordinário comparar o Python de agora com o de cinco anos atrás – a linguagem mudou um pouco, enquanto o mundo no qual ela vive é muito mais amplo e rico. Ele borbulha de entusiasmo, código e idéias (de pessoas espantosamente brilhantes e malucos semelhantes), enquanto permanece respeitoso e cordial. Vamos continuar fazendo isso.

Vancouver, British Columbia, Canadá

Sumário

Parte I Começando

Parte II Tipos e operações

Parte III Instruções e sintaxe

Parte IV Funções

Parte V Módulos

Parte VI Classes e POO

Parte IX Apêndices

PARTE I

Começando

A Parte I começa explorando algumas idéias por trás do Python, o modelo de execução e as maneiras de dar início ao código do programa. Só vamos realmente escrever código na próxima parte, mas certifique-se de ter pelo menos algum conhecimento básico dos objetivos do projeto e das técnicas de execução do Python abordados aqui, antes de passar para os detalhes da linguagem.

1

Uma Sessão de Perguntas e Respostas sobre Python

Se você comprou este livro, talvez já saiba o que é Python e porque é importante conhecer essa ferramenta. Caso contrário, você provavelmente não se convencerá do valor do Python até aprender a linguagem, lendo o restante deste livro e realizando um ou dois projetos. Mas, antes de entrarmos nos detalhes, apresentaremos sucintamente alguns das razões da popularidade do Python. Para começar a moldar uma definição do Python, este capítulo tem a forma de uma sessão de perguntas e respostas, as quais respondem a algumas das questões não-técnicas mais comuns entre os principiantes.

POR QUE AS PESSOAS USAM PYTHON?

Como hoje existem muitas linguagens de programação, essa normalmente é a primeira pergunta dos novatos. Dadas as centenas de milhares de usuários de Python hoje existentes, na verdade não há como responder essa pergunta com total precisão. A escolha das ferramentas de desenvolvimento é às vezes baseada em limitações específicas ou na preferência pessoal.

Mas, após ensinar Python para cerca de 1.000 alunos e em quase 100 empresas nos últimos anos, percebi algumas motivações comuns:

Qualidade do software

Para muitas pessoas, o enfoque que o Python dá à legibilidade, coerência e qualidade do software em geral, o distingue das linguagens estilo "kitchen sink" *, como o Perl. O código em Python é projetado para ser legível e, portanto, fácil de manter – muito mais do que as linguagens de script tradicionais. Além disso, o Python tem excelente suporte para mecanismos de reutilização de código, como a programação orientada a objetos (POO).

---

* N. de R.T.: Linguagens em que a operação é mais importante do que o efeito obtido.

Produtividade do desenvolvedor

O Python aumenta a produtividade do desenvolvedor muitas vezes além do que conseguem as linguagens compiladas ou estaticamente tipadas, como C, C++ e Java. O código em Python normalmente tem de 1/3 a 1/5 do tamanho do código equivalente em C++ ou Java. Isso significa menos digitação, menos depuração e menos manutenção após o desenvolvimento. Os programas em Python também são executados imediatamente, sem as cansativas etapas de compilação e vinculação de algumas outras ferramentas.

Portabilidade do programa

A maioria dos programas em Python é executado sem necessidade de alteração em todas as principais plataformas. Portar um código Python entre o Unix e o Windows, por exemplo, normalmente é apenas uma questão de copiar o código entre as máquinas. Além disso, o Python oferece várias opções para desenvolvimento de interfaces gráficas de usuário portáveis.

Bibliotecas de suporte

O Python vem com um grande conjunto de funcionalidades pré-compiladas e portáveis, conhecidas como bibliotecas padrão. Essas bibliotecas suportam diversas tarefas de programação em nível de aplicativo, desde text pattern matching* até scripts de rede. Além disso, o Python pode ser estendido com bibliotecas feitas em casa, assim como com um enorme conjunto de software de suporte de aplicativo de outros fornecedores.

Integração de componentes

Os scripts em Python podem se comunicar facilmente com outras partes de um aplicativo, usando uma variedade de mecanismos de integração. Essas integrações permitem que o Python seja usado como uma ferramenta de personalização e extensão de produto. Atualmente, o código Python pode chamar bibliotecas C e C++, pode ser chamado a partir de programas em C e C++, pode ser integrado com componentes Java, pode se comunicar por meio de COM, Corba e.NET e pode interagir pela rede, com interfaces como SOAP e XML-RPC.

Prazer

Devido à facilidade de uso e ao conjunto de ferramentas incorporado do Python, ele pode tornar a atividade de programação mais um prazer do que um trabalho. Embora essa possa ser uma vantagem intangível, seu efeito na produtividade de modo geral é um trunfo importante.

Desses fatores, os dois primeiros, qualidade e produtividade, provavelmente são as vantagens mais interessantes para a maioria dos usuários de Python.

Qualidade do software

Por design, o Python implementa uma sintaxe deliberadamente simples e legível, e um modelo de programação altamente coerente. Conforme atesta o slogan de uma recente conferência sobre a linguagem, o resultado é que o Python parece simplesmente "caber em seu cérebro" – isto é, os recursos da linguagem interagem de maneiras consistentes e limitadas, e resultam naturalmente de um pequeno conjunto de conceitos principais. Isso torna a linguagem mais fácil de aprender, entender e lembrar. Na prática, os programadores de Python não precisam referir-se constantemente a manuais ao ler ou escrever código; trata-se de um design ortogonal.

* N. de R.T.: Correspondência de padrão de texto.

Por questão de filosofia, o Python adota uma estratégia um tanto *minimalista*. Isso significa que, embora existam várias maneiras de executar uma tarefa, normalmente há apenas uma maneira óbvia, algumas alternativas menos óbvias e um pequeno conjunto de interações coerentes em toda linguagem. Além disso, o Python não toma decisões arbitrárias; quando as interações são ambíguas, uma intervenção explícita é preferida, em vez de "mágica". Na maneira Python de pensar, explícito é melhor do que implícito e simples é melhor do que complexo.*

Além desses temas de projeto, o Python inclui ferramentas como módulos e POO, que promovem a reutilização de código naturalmente. E como o Python tem como foco a qualidade, os programadores de Python também têm esse foco.

Produtividade do desenvolvedor

Durante a grande explosão da Internet, em meados dos anos 90, era difícil encontrar programadores para implementar projetos de software; os desenvolvedores eram solicitados a implementar sistemas com a mesma rapidez com que a Internet evoluía. Agora, na era pós-explosão, de demissões e recessão econômica, o quadro mudou. Atualmente, as equipes de programação são obrigadas a executar as mesmas tarefas com menos pessoas.

Nesses dois cenários, o Python se distingue como uma ferramenta que permite aos programadores fazer mais com menos esforço. Ela é deliberadamente otimizada para *velocidade de desenvolvimento* – sua sintaxe simples, tipagem dinâmica, ausência de etapas de compilação e seu conjunto de ferramentas incorporado, permitem que os programadores desenvolvam programas em uma fração do tempo de desenvolvimento necessário para algumas outras ferramentas. O resultado é que o Python aumenta a produtividade do desenvolvedor muitas vezes além do que se consegue com as linguagens tradicionais. Essa é uma boa notícia em tempos de crescimento e de recessão.

PYTHON É UMA LINGUAGEM DE SCRIPT?

O Python é uma linguagem de programação de propósito geral, freqüentemente aplicada em funções de script. Ela é comumente definida como uma *linguagem de script orientada a objetos* – uma definição que combina suporte para POO com orientação global voltada para funções de script. Na verdade, as pessoas freqüentemente usam o termo "script", em vez de "programa", para descrever um arquivo de código Python. Neste livro, os termos "script" e "programa" são usados indistintamente, com uma ligeira preferência para "script" para descrever um arquivo de nível superior mais simples e "programa" para se referir a um aplicativo de vários arquivos mais sofisticado.

Como o termo "script" tem significados diferentes para diferentes observadores, alguns preferem que ele não seja aplicado ao Python. Na verdade, as pessoas tendem a pensar em três definições muito diferentes quando ouvem o Python rotulado como uma linguagem de "script", algumas das quais são mais úteis do que outras:

Ferramentas de shell

Ferramentas para desenvolver scripts orientados a sistemas operacionais. Tais programas freqüentemente são executados a partir de linhas de comando no console e realizam ta-

* Para uma visão mais completa da filosofia Python, digite o comando `import this` em qualquer prompt interativo da linguagem (você verá como fazer isso, no Capítulo 2). Isso ativa um "easter egg" oculto no Python, um conjunto de princípios do projeto. Recentemente, o acrônimo EIBTI virou moda para a regra "explícito é melhor do que implícito" (do inglês, Explicit Is Better Than Implicit).

refas como processamento de arquivos de texto e inicialização de outros programas. Os programas em Python podem servir a tais funções, mas esse é apenas um das dezenas de domínios de aplicação do Python. Ele não é apenas uma linguagem de script de shell melhorada.

Linguagem de controle

Uma camada de "cola", usada para controlar e dirigir outros componentes do aplicativo (isto é, o script). Na realidade, os programas em Python freqüentemente são distribuídos no contexto de um aplicativo maior. Por exemplo, para testar dispositivos de hardware, os programas em Python podem chamar componentes que dão acesso de baixo nível para um dispositivo. Analogamente, os programas podem executar trechos de código Python em pontos estratégicos, para suportar personalização de produto para o usuário final, sem ter de distribuir e recompilar o código-fonte do sistema inteiro. A simplicidade do Python o torna uma ferramenta de controle naturalmente flexível. Tecnicamente, contudo, essa também é apenas uma função comum do Python; muitos programadores de Python desenvolvem scripts independentes sem nem mesmo usar ou ter conhecimento sobre quaisquer componentes integrados.

Facilidade de uso

Uma linguagem simples, usada para completar tarefas rapidamente. Essa provavelmente é a melhor maneira de pensar no Python como uma linguagem de script. O Python permite que os programas sejam desenvolvidos muito mais rapidamente do que nas linguagens compiladas, como C++. Seu rápido ciclo de desenvolvimento promove um modo de programação exploratório e incremental que precisa ser experimentado para ser apreciado. Contudo, não se engane – o Python não serve apenas para tarefas simples. Pelo contrário, ele torna as tarefas simples, por sua facilidade de uso e flexibilidade. O Python tem um conjunto de recursos simples, mas permite que os programas se tornem cada vez mais sofisticados, conforme for necessário.

Então, o Python é uma linguagem de script ou não? Isso depende de para quem você pergunta. Em geral, o termo script provavelmente é melhor usado para descrever o modo de desenvolvimento rápido e flexível que o Python suporta, em vez de um domínio de aplicação em particular.

CERTO, MAS QUAL É O INCONVENIENTE?

Talvez o único inconveniente do Python seja que, conforme a implementação, sua velocidade de execução pode nem sempre ser tão rápida quanto as linguagens compiladas, como C e C++.

Vamos falar sobre conceitos de implementação posteriormente neste livro, mas, em resumo, as implementações padrão do Python atualmente compilam (isto é, traduzem) instruções do código-fonte em um formato intermediário conhecido como *código de byte* e, depois, interpretam o código de byte. O código de byte proporciona portabilidade, pois é um formato independente de plataforma. Entretanto, como o Python não é compilado até se tornar código de máquina binário (por exemplo, instruções para um chip Intel), alguns programas serão executados mais lentamente em Python do que em uma linguagem totalmente compilada, como C.

Se você vai ou não se *preocupar* com a diferença na velocidade de execução, depende dos tipos de programa que vai escrever. O Python foi otimizado várias vezes e o código Python sozinho é executado rápido o bastante na maioria dos domínios de aplicação. Além disso, quando você faz algo "real" em um script Python, como processar um arquivo ou construir uma GUI, seu programa está na realidade executando na velocidade da linguagem C, pois

essas tarefas são enviadas imediatamente para código C compilado, dentro do interpretador Python. Mais fundamentalmente, o ganho na velocidade de desenvolvimento do Python é freqüentemente bem mais importante do que qualquer perda na velocidade de execução, especialmente em face da velocidade dos computadores modernos.

Mesmo com as velocidades de CPU atuais, ainda existem alguns domínios que exigem uma velocidade de execução otimizada. A programação numérica e a animação, por exemplo, freqüentemente precisam de que pelo menos seus componentes básicos de tratamento numérico sejam executados na velocidade da linguagem C (ou mais rápido). Se você trabalha em um domínio assim, ainda pode usar o Python – basta separar as partes do aplicativo que exigem velocidade ótima em *extensões compiladas* e vinculá-las em seu sistema para usar nos scripts Python.

Não vamos falar muito sobre extensões neste texto, pois esse é apenas um exemplo da função do Python como linguagem de controle, que discutimos anteriormente. Um ótimo exemplo dessa estratégia de linguagem dupla é a extensão de programação numérica *NumPy* do Python; *combinando* bibliotecas de extensão numérica otimizadas e compiladas com a linguagem Python, a extensão NumPy transforma o Python em uma ferramenta de programação numérica eficiente e fácil de usar. Talvez você nunca precise codificar tais extensões em seu trabalho com o Python, mas elas fornecem um mecanismo de otimização poderoso, se precisar.

QUEM USA PYTHON ATUALMENTE?

Quando este livro estava em produção, em 2003, a melhor estimativa que alguém poderia fazer para o tamanho da base de usuários do Python situava-se entre 500.000 e 1 milhão de usuários em todo o mundo. Essa estimativa foi baseada em várias estatísticas, como downloads e tráfego de newsgroup comparativo. Como o Python tem código-fonte aberto, é difícil obter uma contagem mais exata – não há nenhum registro de licenciamento para calcular. Além disso, o Python é incluído automaticamente nas distribuições de Linux e de alguns produtos e hardware de computador, obscurecendo ainda mais o quadro da base de usuários. Em geral, contudo, o Python ostenta uma base de usuários grande e uma comunidade de desenvolvedores muito ativa. Como o Python está no mercado há mais de uma década e é muito usado, ele também é estável e robusto.

Além dos usuários individuais, o Python também é aplicado em produtos de geração de renda reais, por empresas reais. Por exemplo, Google e Yahoo! utilizam Python em serviços de Internet; a Hewlett-Packard, a Seagate e a IBM usam Python para testes de hardware; a Industrial Light and Magic e outras empresas usam Python na produção de animação de filmes; e assim por diante. Provavelmente, o único ponto em comum entre as empresas que usam Python atualmente é que ele é utilizado em todo o espectro, em termos de domínios de aplicação. Sua natureza de propósito geral o torna aplicável em quase todos os campos e não apenas em um. Para obter mais detalhes sobre as empresas que usam Python atualmente, consulte o site da Web do Python, no endereço *http://www.python.org*.

O QUE EU POSSO FAZER COM PYTHON?

Além de ser uma linguagem de programação bem projetada, o Python também é útil para executar tarefas do mundo real – os tipos de coisas que os desenvolvedores fazem todo dia. Ele é comumente usado em diversos domínios, como ferramenta para escrever outros componentes e implementar programas independentes. Na verdade, como linguagem de propósito geral, as funções do Python são praticamente ilimitadas.

Entretanto, as funções mais comuns do Python parecem recair em algumas categorias mais amplas. As próximas seções descrevem algumas das aplicações mais comuns do Python, assim como as ferramentas usadas em cada domínio. Não podemos descrever todas as ferramentas mencionadas aqui; se você estiver interessado em algum desses assuntos, veja o Python on-line ou outros recursos, para obter mais detalhes.

Programação de sistemas

As interfaces incorporadas do Python para serviços de sistema operacional o tornam ideal para escrever ferramentas e utilitários de administração de sistemas portáveis e fáceis de manter (às vezes chamadas de ferramentas de shell). Os programas em Python podem pesquisar arquivos e árvores de diretório, chamar outros programas, realizar processamento paralelo com processos e segmentos etc.

A biblioteca padrão do Python vem com vínculos POSIX e suporte para todas as ferramentas de sistema operacional comuns: variáveis de ambiente, arquivos, sockets, pipes, processos, múltiplos segmentos, expressões regulares para correspondência de padrão de texto, argumentos de linha de comando, interfaces de fluxo padrão, execução de comando de shell, expansão de nome de arquivo e muito mais. Além disso, a maior parte das interfaces de sistema do Python é projetada para ser portável; por exemplo, um script que copia árvores de diretório normalmente é executado sem alteração em todas as principais plataformas Python.

GUIs

A simplicidade e o rápido retorno do Python também o tornam bom para programação de GUI (interface gráfica com o usuário). O Python vem com uma interface orientada a objetos padrão para a API de GUI Tk, chamada Tkinter, que permite aos programas em Python implementarem GUIs portáveis com aparência e comportamento nativos. As GUIs Python/Tkinter são executadas sem alteração no MS Windows, X Windows (no Unix e no Linux) e em Macs. Um pacote de extensões gratuito, o PMW, acrescenta elementos de janela avançados no kit de ferramentas Tkinter básico. Além disso, a API de GUI wxPython, baseada em uma biblioteca C++, oferece um kit de ferramentas alternativo para a construção de GUIs portáveis em Python.

Kits de ferramentas de nível mais alto, como *PythonCard* e *PMW*, são construídos em cima das APIs básicas, como wxPython e Tkinter. Com a biblioteca apropriada, você também pode usar outros kits de ferramentas de GUI no Python, como Qt, GTK, MFC e Swing. Para aplicativos executados em navegadores da Web ou que possuem requisitos de interface simples, os scripts CGI no lado do servidor Jython e Python oferecem opções adicionais de interface com o usuário.

Scripts de Internet

O Python vem com módulos para Internet padrão que permitem aos programas executar uma grande variedade de tarefas em rede, tanto no modo cliente como servidor. Os scripts podem comunicar-se por meio de sockets; extrair informações de formulários enviados para um script CGI no lado do servidor; transferir arquivos por meio de FTP; processar arquivos XML; enviar, receber e analisar email; buscar páginas da Web por meio de URLs; analisar o código HTML e XML das páginas buscadas da Web; comunicar-se por meio de XML-RPC, SOAP e telnet; e muito mais. As bibliotecas Python tornam essas tarefas notavelmente simples.

Além disso, há um grande conjunto de ferramentas de outros fornecedores na Web para fazer programação para Internet em Python. Por exemplo, o sistema *HTMLGen* gera arquivos HTML a partir de descrições baseadas em classes Python; o pacote de extensões Windows *win32all* permite que o código Python seja incorporado em arquivos HTML como se fosse JavaScript; o pacote mod_python executa código Python eficientemente dentro do servidor da Web Apache; e o sistema Jython fornece integração Python/Java transparente e suporta codificação de applets no lado do servidor que são executados nos clientes. Além disso, pacotes de desenvolvimento da Web completos para Python, como Zope, WebWare e Quixote, suportam a construção rápida de sites da Web.

Integração de componentes

Discutimos anteriormente a função da integração de componentes, ao descrevermos o Python como uma linguagem de controle. A capacidade do Python de ser estendido e incorporado em sistemas C e C++ o torna útil como uma linguagem de "cola" flexível para fazer o script de controle de comportamento de outros sistemas e componentes. Por exemplo, integrando uma biblioteca C no Python, este pode testar e ativar seus componentes. E incorporando o Python em um produto, personalizações locais podem ser desenvolvidas sem a necessidade de recompilar o produto inteiro nem de distribuir seu código-fonte.

Ferramentas como o gerador de código SWIG podem automatizar grande parte do trabalho necessário para vincular componentes compilados no Python, para uso em scripts. E estruturas maiores, como o suporte para COM do Python para o MS Windows, a implementação do Jython baseado em Java, o sistema Python.NET e vários kits de ferramentas CORBA para o Python, oferecem maneiras alternativas para fazer o script de componentes. No Windows, por exemplo, os scripts Python podem usar estruturas para script do MS Word e do Excel, e proporcionar os mesmos tipos de funções do Visual Basic.

Programação de banco de dados

O módulo pickle padrão do Python fornece um sistema de *persistência de objeto* simples – ele permite que os programas salvem e restaurem facilmente objetos Python inteiros em arquivos e objetos do tipo arquivo. Para demandas de banco de dados mais tradicionais, existem interfaces Python para Sysbase, Oracle, Informix, ODBC, MySQL e muito mais.

O mundo Python também definiu uma *API de banco de dados portável* para acessar sistemas de banco de dados SQL a partir de scripts Python, a qual tem a mesma aparência em diversos sistemas de banco de dados subjacentes. Por exemplo, como as interfaces de fornecedor implementam a API portável, um script escrito para trabalhar com o sistema gratuito MySQL funcionará praticamente sem alteração em outros sistemas, como o Oracle, simplesmente substituindo-se a interface do fornecedor subjacente. Na Web, você também encontrará um sistema de outro fornecedor, chamado *gadfly*, que implementa um banco de dados SQL para programas em Python, e um sistema de banco de dados completo orientado a objetos, chamado *ZODB*.

Composição rápida de protótipos

Para os programas em Python, os componentes escritos em Python e C parecem iguais. Por isso, é possível fazer o protótipo de sistemas inicialmente em Python e, então, mover componentes para uma linguagem compilada, como C ou C++, para distribuição. Ao contrário de algumas ferramentas de produção de protótipos, o Python não exige uma reescrita completa,

uma vez que o protótipo tenha se solidificado. As partes do sistema que não exigem a eficiência de uma linguagem como C++ podem permanecer escritas em Python para facilidade de manutenção e uso.

Programação numérica

A extensão de programação numérica *NumPy* do Python, mencionada anteriormente, inclui ferramentas avançadas, como um objeto array, interfaces para bibliotecas matemáticas padrão e muito mais. Integrando o Python com rotinas numéricas escritas em uma linguagem compilada, para obter velocidade, a extensão NumPy transforma o Python em uma ferramenta de programação numérica sofisticada, porém fácil de usar, que freqüentemente pode substituir código já existente escrito em linguagens compiladas tradicionais, como FORTRAN ou C++. Ferramentas numéricas adicionais para Python suportam animação, visualização em 3D etc.

Jogos, imagens, IA, XML e muito mais

O Python é comumente aplicado em mais domínios do que podemos mencionar aqui. Por exemplo, você pode fazer programação gráfica e de jogos em Python, com o sistema *pygame*; processamento de imagens, com o pacote *PIL* e outros; programação de IA com simuladores de redes neurais e shells de sistemas especialistas; análise de código XML com o pacote de bibliotecas `xml`, o módulo `xmlrpclib` e extensões de outros fornecedores; e até jogar paciência com o programa *PySol*. Você encontrará suporte para muitas dessas áreas no site da Web Vaults of Parnassus (com link a partir de *http://www.python.org*). (O Vaults of Parnassus é uma grande coleção de links para software de outros fornecedores para programação em Python. Se você precisa fazer algo especial com o Python, o Vaults normalmente é o melhor lugar para procurar recursos.)

Em geral, muitos desses domínios específicos são, em grande parte, apenas exemplos da função de integração de componentes do Python novamente em ação. Adicionando o Python como um frontend para bibliotecas de componentes escritos em uma linguagem compilada, como C, o Python se torna útil para scripts de uma ampla variedade de domínios. Como uma ferramenta de propósito geral que suporta integração, o Python é amplamente aplicável.

QUAIS SÃO AS VANTAGENS TÉCNICAS DO PYTHON?

Naturalmente, essa é uma pergunta de um desenvolvedor. Se você ainda não tem experiência em programação, o que diremos nas próximas seções pode ser um pouco desconcertante – não se preocupe, explicaremos tudo isso com mais detalhes à medida que avançarmos neste livro. Para o desenvolvedor, contudo, esta é uma rápida introdução a alguns dos principais recursos técnicos do Python.

É orientado a objetos

O Python é uma linguagem completamente orientada a objetos. Seu modelo de classes suporta noções avançadas, como polimorfismo, sobrecarga de operadores e herança múltipla; contudo, no contexto da sintaxe e tipagem simples do Python, a POO é notadamente fácil de aplicar. Na verdade, se você não entende esses termos, verá que eles são muito mais fáceis de aprender com o Python do que com praticamente qualquer outra linguagem de POO disponível.

Além de servir como um poderoso dispositivo de estruturação e reutilização de código, a natureza POO do Python o torna ideal como ferramenta de script para linguagens de sistemas orientados a objetos, como C++ e Java. Por exemplo, com o código de "cola" apropriado, os

programas em Python podem transformar as classes implementadas em C++ ou Java em subclasses (especializá-las). De igual importância é o fato de que a POO é uma *opção* no Python; você pode ir longe sem precisar se tornar ao mesmo tempo um especialista em objetos.

É gratuito

O Python é gratuito. Assim como qualquer outro software de código-fonte aberto, como Tcl, Perl, Linux e Apache, você pode obter o sistema Python inteiro gratuitamente na Internet. Não existe nenhuma restrição para cópia, incorporação em seus sistemas ou distribuição com seus produtos. Na verdade, você pode até vender o código-fonte do Python, se tiver esse perfil.

Mas não entenda errado: "gratuito" não significa "sem suporte". Pelo contrário, a comunidade on-line do Python responde às consultas dos usuários com uma velocidade que a maioria dos fornecedores de software comerciais faria bem em notar. Além disso, como o Python vem com o código-fonte completo, ele dá poderes aos desenvolvedores e cria uma grande equipe de especialistas em implementação. Embora estudar ou alterar a implementação de uma linguagem de programação não seja a noção de divertimento de todo mundo, é reconfortante saber que ela está disponível como último recurso e como fonte de documentação definitiva. Você não fica dependente de um fornecedor comercial.

O desenvolvimento do Python é realizado por uma comunidade, a qual coordena, em grande parte, seus esforços pela Internet. Ela é composta pelo criador do Python – Guido van Rossum, o oficialmente ungido Ditador Benevolente da Vida (DBDV) do Python – e mais milhares de pessoas.

As alterações na linguagem devem seguir um procedimento de aprimoramento formal (conhecido como processo PEP) e serem minuciosamente examinadas pelo DBDV. Felizmente, isso tende a tornar o Python mais conservador com relação a alterações do que algumas outras linguagens.

É portável

A implementação padrão do Python é escrita em ANSI C portável, compila e executa em praticamente todas as principais plataformas em uso atualmente. Por exemplo, os programas em Python são executados em tudo, de PDAs até supercomputadores. Como uma lista parcial, o Python está disponível em sistemas Unix, Linux, MS-DOS, MS Windows (95, 98, NT, 2000, XP etc.), Macintosh (classic e OS X), Amiga, AtariST, BeOS, OS/2, VMS, QNX, Vxworks, PalmOS, PocketPC e Windows CE, supercomputadores Cray, computadores de grande porte IBM, PDAs executando Linux e muito mais.

Além do interpretador de linguagem em si, o conjunto de módulos de bibliotecas padrão que acompanha o Python também é implementado para ser portável o máximo possível entre plataformas. Além disso, os programas em Python são compilados automaticamente em código de byte portável, o qual é executado igualmente em qualquer plataforma com uma versão de Python compatível instalada (mais informações sobre isso no próximo capítulo).

Isso significa que os programas em Python que usam a linguagem básica e as bibliotecas padrão são executados igualmente em sistemas Unix, MS Windows e na maioria dos outros, com um interpretador Python. A maior parte dos ports* Python também contém extensões específicas da plataforma (por exemplo, suporte para COM no MS Windows), mas a lin-

* N. de R.T.: Sistema de instalação de software do BSD, como RPM e APT para Linux.

guagem Python básica e as bibliotecas funcionam igualmente em qualquer lugar. Conforme mencionado anteriormente, o Python também inclui uma interface para o kit de ferramentas de GUI Tk, chamada Tkinter, a qual permite que os programas implementem interfaces gráficas de usuário completas, que são executadas em todas as principais plataformas de GUI sem alterações no programa.

É poderoso

Da perspectiva de recursos, o Python é um pouco híbrido. Seu conjunto de ferramentas o coloca entre as linguagens de script tradicionais (como Tcl, Scheme e Perl) e as linguagens de desenvolvimento de sistemas (como C, C++ e Java). O Python oferece toda a simplicidade e facilidade de uso de uma linguagem de script, junto com ferramentas de engenharia de software mais avançadas, normalmente encontradas nas linguagens compiladas. Ao contrário de algumas linguagens de script, essa combinação torna o Python útil para projetos de desenvolvimento de grande escala. Como uma prévia, aqui estão alguns dos principais itens que você encontrará nas ferramentas do Python:

Tipagem dinâmica

O Python monitora os tipos de objetos que seu programa utiliza ao executar; ele não exige complicadas declarações de tipo e tamanho em seu código. Na verdade, conforme veremos no Capítulo 4, não existe algo como declaração de tipo ou variável em qualquer parte no Python.

Gerenciamento de memória automático

O Python aloca e recupera automaticamente os objetos ("coletas de lixo"), quando não são mais usados, e a maioria cresce ou diminui sob demanda. O Python monitora os detalhes da memória de baixo nível, para que você não precise fazer isso.

Suporte para programação em grande escala

Para a construção de sistemas maiores, o Python inclui ferramentas como módulos, classes e exceções. Essas ferramentas permitem que você organize os sistemas em componentes, utilize POO para reutilizar e personalizar código, e trate de eventos e erros de modo elegante.

Tipos de objeto incorporados

O Python fornece as estruturas de dados comumente usadas, como listas, dicionários e strings, como uma parte intrínseca da linguagem; como veremos, elas são flexíveis e fáceis de usar. Por exemplo, os objetos incorporados podem crescer e diminuir sob demanda, podem ser aninhados arbitrariamente para representar informações complexas e muito mais.

Ferramentas incorporadas

Para processar todos esses tipos de objetos, o Python vem com poderosas operações padrão, incluindo concatenação (junção de coleções), fracionamento (extração de seções), ordenação, mapeamento e muito mais.

Bibliotecas

Para tarefas mais específicas, o Python também vem com um grande conjunto de bibliotecas previamente desenvolvidas, que suportam tudo, desde correspondência através de expressões regulares até comunicação em rede. As bibliotecas do Python estão no local onde ocorre grande parte das ações em nível de aplicativo.

Utilitários de outros fornecedores

Como o Python é freeware, ele estimula os desenvolvedores a contribuírem com ferramentas previamente desenvolvidas que suportam tarefas além das incorporadas na linguagem; você encontrará suporte gratuito para COM, geração de imagens, ORBs CORBA, XML, fornecedores de banco de dados e muito mais.

Apesar da variedade de ferramentas no Python, ele mantém uma sintaxe e um design notadamente simples. O resultado é uma ferramenta de programação poderosa que mantém a capacidade de utilização de uma linguagem de script.

Pode ser misturado

Os programas em Python podem ser facilmente "colados" em componentes escritos em outras linguagens, de diversas maneiras. Por exemplo, a API C do Python permite que programas em C chamem e sejam chamados por programas em Python, de forma flexível. Isso significa que você pode adicionar funcionalidade no sistema Python, quando necessário, e usar os programas em Python dentro de outros ambientes ou sistemas.

Por exemplo, misturando Python com bibliotecas desenvolvidas em linguagens como C ou C++, ele se torna uma linguagem de frontend e uma ferramenta de personalização fácil de usar. Conforme mencionado anteriormente, isso também torna o Python bom na produção rápida de protótipos; os sistemas podem ser primeiro implementados em Python para aumentar sua velocidade de desenvolvimento e, posteriormente, serem movidos para C para distribuição, uma parte por vez, de acordo com as exigências de desempenho.

É fácil de usar

Para executar um programa em Python, você simplesmente o digita e executa. Não há etapas de compilação e ligação intermediárias, como acontece em linguagens como C ou C++. O Python executa os programas imediatamente, o que favorece uma experiência de programação interativa e um retorno rápido, após alterações no programa.

É claro que o ciclo de desenvolvimento é apenas um aspecto da facilidade de uso do Python. Ele também fornece uma sintaxe deliberadamente simples e poderosas ferramentas de alto nível incorporadas. Na verdade, alguns chegam ao ponto de chamar o Python de "pseudocódigo executável". Como ele elimina grande parte da complexidade existente em outras ferramentas, os programas em Python são mais simples, menores e mais flexíveis do que os programas equivalentes em linguagens como C, C++ e Java.

É fácil de aprender

Isso nos leva ao assunto deste livro: comparada com outras linguagens de programação, a linguagem Python básica é muito fácil de aprender. Na verdade, você poderá estar desenvolvendo programas significativos em Python em questão de dias (e talvez em apenas algumas horas, se já é um programador experiente). Essa é uma boa notícia, tanto para desenvolvedores profissionais que estejam procurando aprender a linguagem para usar no trabalho, como para usuários finais de sistemas que disponibilizam uma camada em Python para personalização e controle. Atualmente, muitos sistemas contam com o fato de os usuários finais poderem aprender rapidamente sobre Python para mexerem em seus códigos de personalização no local, com pouco ou nenhum suporte.

EM QUE O PYTHON É MELHOR DO QUE A LINGUAGEM X?

Às vezes as pessoas comparam o Python com linguagens como Perl, Tcl e Java. Falamos sobre o desempenho anteriormente; portanto, aqui, o foco é a funcionalidade. Embora outras linguagens também sejam ferramentas úteis, achamos que o Python:

- É mais poderoso do que Tcl. O suporte do Python para "programação em grande escala" o torna aplicável para desenvolvimento de sistemas maiores.
- Tem uma sintaxe mais limpa e um design mais simples do que Perl, o que o torna mais legível e fácil de manter, e ajuda a reduzir erros de programa.
- É mais simples de usar do que Java. O Python é uma linguagem de script, mas a Java herda grande parte da complexidade das linguagens de sistema, como C++.
- É mais simples e fácil de usar do que a C++, e quase não compete com esta; como uma linguagem de script, muitas vezes o Python tem funções diferentes.
- É mais poderoso e mais independente de plataforma do que o Visual Basic. Sua natureza de código-fonte aberto também significa que ele não é controlado por uma única empresa.
- Tem a característica dinâmica de linguagens como SmallTalk e Lisp, mas também tem uma sintaxe tradicional simples, acessível para desenvolvedores e usuários finais.

Especialmente para programas que fazem mais do que varrer arquivos de texto e que talvez tenham de ser lidos por outras pessoas no futuro (ou por você!), nós achamos que o Python é mais adequado do que qualquer outra linguagem de script disponível atualmente. Além disso, a menos que seu aplicativo exija o máximo desempenho, o Python é uma alternativa viável às linguagens de desenvolvimento de sistemas, como C, C++ e Java; o código Python será muito menos difícil de escrever, depurar e manter.

É claro que os dois autores são defensores de carteirinha do Python; portanto, receba estes comentários como quiser. Contudo, eles refletem a experiência comum de muitos desenvolvedores que tiveram tempo para explorar o que o Python tem a oferecer.

E isso conclui a parte de propaganda deste livro. A melhor maneira de julgar uma linguagem é vê-la em ação; portanto, os dois próximos capítulos apresentam uma introdução rigorosamente técnica da linguagem. Lá, exploraremos as maneiras de executar programas em Python, examinaremos seu modelo de execução de código de byte e apresentaremos os fundamentos dos arquivos modulares para salvar seu código. Nosso objetivo é fornecer informações suficientes para você executar os exemplos e exercícios do restante do livro. Como já dissemos, você não começará a programar realmente até o Capítulo 4, mas examine os detalhes iniciais, antes de prosseguir.

2

Como o Python Executa Programas

Este e o próximo capítulos fornecem uma rápida visão da execução de programas – como você dá início ao código e como o Python o executa. Neste capítulo, explicamos o interpretador Python. O Capítulo 3 mostrará como você faz para preparar e executar seus próprios programas.

Os detalhes da inicialização são inerentemente específicos da plataforma e parte do material deste capítulo pode não se aplicar à plataforma na qual você está trabalhando; portanto, você deve se sentir à vontade para pular as partes que não sejam relevantes para seu uso planejado. Na verdade, os leitores mais avançados, que tenham usado ferramentas semelhantes no passado e prefiram chegar ao cerne da linguagem rapidamente, talvez queiram deixar parte deste capítulo para referência futura. Para o restante, vamos aprender como se executa um código.

APRESENTANDO O INTERPRETADOR PYTHON

Até aqui, estivemos falando principalmente sobre Python como linguagem de programação. Mas, após implementado, ele também é um pacote de software chamado de *interpretador*. Um interpretador é um tipo de programa que executa outros programas. Quando você escreve programas em Python, o interpretador lê seu programa e executa as instruções que ele contém. Na verdade, o interpretador é uma camada de software lógico entre seu código e o hardware do computador em sua máquina.

Quando o pacote Python é instalado em sua máquina, ele gera vários componentes – no mínimo, um interpretador e uma biblioteca de suporte. Dependendo de como você o utiliza, o interpretador Python pode assumir a forma de um programa executável ou de um conjunto de bibliotecas vinculadas a outro programa. Dependendo do tipo de Python que você executa, o próprio interpretador pode ser implementado como um programa em C, como um conjunto de classes Java ou outro. Qualquer que seja a forma assumida, o código Python que você escreve deve ser sempre executado por esse interpretador. E, para fazer isso, você deve primeiro instalar um interpretador Python em seu computador.

Os detalhes da instalação do Python variam de acordo com a plataforma e são abordados em detalhes no Apêndice A. Em resumo:

- Os usuários de Windows fazem o download e executam um arquivo executável de instalação automática, o qual coloca o Python em suas máquinas. Basta dar um clique duplo e responder Sim ou Avançar em todos os prompts.
- Os usuários de Linux e Unix normalmente instalam o Python a partir de arquivos RPM ou o compilam a partir de seu pacote de distribuição de código-fonte.
- Outras plataformas têm técnicas de instalação específicas para elas. Por exemplo, no Palm Pilot os arquivos são sincronizados.

O Python pode ser obtido a partir da página de downloads no seu site da Web, no endereço *www.python.org*. Ele também pode ser encontrado por meio de vários outros canais de distribuição. Talvez você já tenha o Python disponível em sua máquina, especialmente no Linux e no Unix. Se você estiver trabalhando no Windows, normalmente encontrará o Python no menu Iniciar, conforme mostrado na Figura 2-1 (aprenderemos o que esses itens de menu significam em breve). No Unix e no Linux, o Python provavelmente fica no */usr*, em sua árvore de diretório.

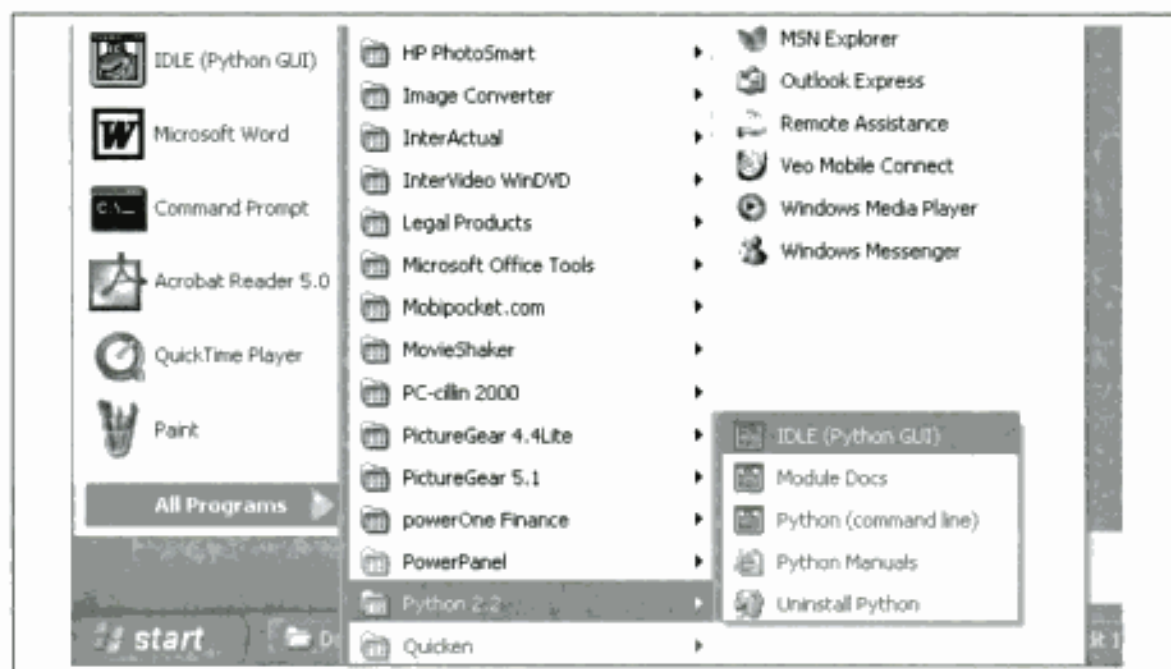



Figura 2-1 Python no menu Iniciar do Windows.

Como os detalhes da instalação são muito específicos, de acordo com o tipo de plataforma, vamos terminar esta explicação por aqui (para ver mais detalhes sobre o processo de instalação, consulte o Apêndice A). Para os propósitos deste capítulo e do próximo, vamos supor que você tenha o Python pronto para usar.

EXECUÇÃO DE PROGRAMA

O significado de escrever e executar um script em Python depende de você estar vendo essas tarefas como programador ou como interpretador do Python. As duas visões oferecem uma perspectiva importante sobre a programação em Python.

A visão do programador

Em sua forma mais simples, um programa em Python é apenas um arquivo de texto contendo instruções nessa linguagem. Por exemplo, o arquivo a seguir, chamado *script1.py*, é um dos scripts em Python mais simples que podemos conceber, mas ele passa por um programa oficial:

```
print 'hello world'
print 2 ** 100
```

Esse arquivo contém duas instruções `print` em Python, as quais simplesmente imprimem uma string (o texto entre apóstrofos) e o resultado de uma expressão numérica (2 elevado à potência 100) no fluxo de saída. Não se preocupe com a sintaxe desse código ainda – para este capítulo, estamos interessados apenas em fazê-lo executar. Vamos explicar a instrução `print` e porque você pode elevar 2 à potência de 100 em Python, sem estouro de pilha, em partes posteriores deste livro.

Você pode criar esse arquivo de instruções com qualquer editor de textos que queira. Por convenção, os arquivos de programa Python recebem nomes que terminam com ".py"; tecnicamente, esse esquema de atribuição de nomes é exigido apenas para arquivos que são "importados", conforme mostrado posteriormente neste livro, mas a maioria dos arquivos Python tem nomes.*py* por consistência.

Após ter digitado essas instruções em um arquivo de texto de uma maneira ou outra, você deve dizer ao Python para que *execute* o arquivo – o que significa simplesmente executar todas as instruções, do início ao fim do arquivo, uma após a outra. Os arquivos de programa Python podem ser iniciados por linhas de comando, por um clique em seus ícones e com outras técnicas padrão. Demonstraremos como se faz para ativar essa execução no próximo capítulo. Se tudo correr bem, você verá o resultado das duas instruções `print` aparecer em algum lugar em seu computador – por padrão, normalmente na mesma janela em que você estava quando executou o programa:

```
hello world
1267650600228229401496703205376
```

Por exemplo, aqui está como esse script foi executado a partir de uma linha de comando do DOS em um laptop Windows, para garantir que ele não tivesse nenhum erro comum de digitação:

```
D:\temp>python script1.py
hello world
1267650600228229401496703205376
```

Acabamos de executar um script Python que imprime uma string e um número. Provavelmente, não ganharemos nenhum prêmio de programação com esse código, mas é suficiente para se entender a base de execução de programas.

Visão do Python

A breve descrição da seção anterior é claramente padrão para linguagens de script, e normalmente é tudo que a maioria dos programadores de Python precisa saber. Você digita código em arquivos de texto e executa esses arquivos por meio do interpretador. Nos bastidores, contudo, um pouco mais acontece quando você diz ao Python para "prosseguir". Embora o conhecimento do funcionamento interno do Python não seja rigorosamente exigido para

programação nessa linguagem, conhecimento básico da estrutura de tempo de execução do Python pode ajudá-lo a compreender o quadro geral da execução de programas.

Quando você instrui o Python para executar seu script, existem algumas etapas que ele executa, antes que seu código comece realmente a funcionar. Especificamente, ele é primeiro compilado para algo chamado código de byte e, depois, é levado a algo chamado máquina virtual.

Compilação de código de byte

Internamente e quase completamente oculto de você, o Python primeiro compila seu *código-fonte* (as instruções presentes em seu arquivo) para um formato conhecido como *código de byte*. A compilação é simplesmente uma etapa de tradução e código de byte é uma representação de nível mais baixo e independente da plataforma do seu código-fonte. A grosso modo, cada uma das suas instruções de código-fonte é transformada em um grupo de instruções em código de byte. Essa transformação em código de byte é realizada para acelerar a execução – o código de byte pode ser executado muito mais rapidamente do que as instruções originais do código-fonte.

Você notará que o parágrafo anterior disse que isso é *quase* completamente oculto. Se o processo do Python tiver acesso de gravação em sua máquina, ele armazenará o código de byte de seu programa em arquivos que terminam com a extensão *.pyc* (".pyc" significa código-fonte ".py" compilado). Você verá esses arquivos aparecerem em seu computador após ter executado os programas. O Python salva códigos de byte como esse para uma otimização da velocidade na inicialização. Na próxima vez que você executar seu programa, o Python carregará o arquivo *.pyc* e pulará a etapa de compilação, contanto que você não tenha alterado seu código-fonte desde que o código de byte foi salvo. O Python verifica automaticamente as datas de alteração dos arquivos de código-fonte e de código de byte, para saber quando deve recompilar.

Se o Python não puder gravar os arquivos de código de byte em sua máquina, seu programa ainda assim funcionará – o código de byte é gerado na memória e simplesmente descartado na saída do programa.* Entretanto, como os arquivos *.pyc* aceleram o tempo de inicialização, você desejará certificar-se que, para programas maiores, eles sejam gravados. Os arquivos de código de byte também são uma maneira de distribuir programas em Python – o Python prontamente executa um programa se tudo o que encontra são arquivos *.pyc*, mesmo que os arquivos-fonte *.py* originais estejam ausentes. (Para outra opção de distribuição, veja a seção "Binários congelados", posteriormente neste capítulo.)

Máquina virtual Python (PVM)

Uma vez que seu programa tenha sido compilado em código de byte (ou que o código de byte tenha sido carregado a partir de arquivos *.pyc*), ele é levado para execução em algo geralmente conhecido como Máquina Virtual Python (PVM – do original em inglês Python Virtual Machine –, para os que gostam de acrônimos). A PVM parece mais impressionante do que é; na realidade, trata-se apenas de um grande loop que faz repetições nas instruções em código de byte, uma por uma, para executar suas operações. A PVM é o mecanismo de

* E, rigorosamente falando, o código de byte é salvo apenas para arquivos que são importados e não para o arquivo de nível superior de um programa. Vamos explorar as importações no Capítulo 3 e, novamente, na Parte V. O código de byte também nunca é salvo para código digitado no prompt interativo, que será descrito no Capítulo 3.

tempo de execução do Python; ela está sempre presente como parte do sistema Python e é o componente que realmente executa seus scripts. Tecnicamente, essa é apenas a última etapa do que é chamado de interpretador Python.

A Figura 2-2 ilustra a estrutura de tempo de execução descrita. Lembre-se de que toda essa complexidade é deliberadamente oculta para programadores de Python. A compilação do código de byte é automática e a PVM é apenas parte do sistema Python que você instalou em sua máquina. Novamente, os programadores simplesmente codificam e executam arquivos de instruções.

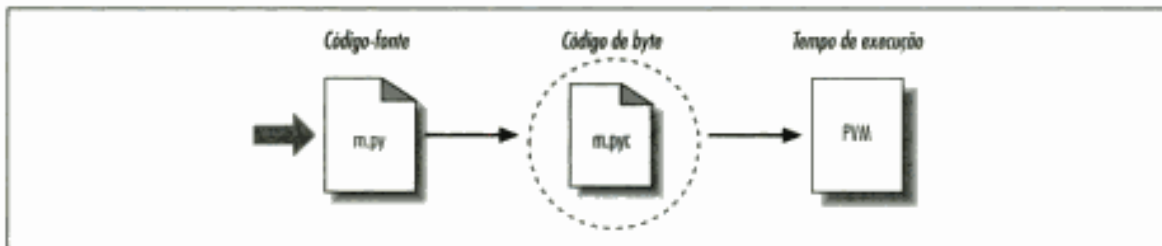



Figura 2-2 Modelo de execução de runtime.

Implicações no desempenho

Os leitores com experiência em linguagens totalmente compiladas, como C e C++, podem notar algumas diferenças no modelo Python. Por exemplo, normalmente não existe nenhuma etapa de construção ou "make" no funcionamento do Python: o código é executado imediatamente após ser escrito. Outro exemplo é que o código de byte Python não é código de máquina binário (como instruções para um chip Intel). O código de byte é uma representação específica do Python.

É por isso que alguns programas desenvolvidos em Python podem não ser executados com a mesma rapidez que na linguagem C ou C++, conforme descrito no Capítulo 1 – o loop PVM (e não o chip da CPU) ainda precisa interpretar o código de byte, e a instruções em código de byte exigem mais trabalho do que as instruções de CPU. Por outro lado, ao contrário dos interpretadores clássicos, internamente há ainda uma etapa de compilação – o Python não precisa analisar e investigar novamente cada instrução do código-fonte repetidamente. O resultado é que o código Python puro é executado em algum lugar entre uma linguagem compilada tradicional e uma linguagem interpretada tradicional. Consulte o Capítulo 1 para mais informações sobre o desempenho do Python.

Implicações no desenvolvimento

Outra ramificação do modelo de execução do Python é que, na realidade, não há nenhuma distinção entre os ambientes de desenvolvimento e execução. Isto é, os sistemas que compilam e executam seu código-fonte são, na verdade, um só. Essa semelhança pode ser mais significativa para os leitores com conhecimento das linguagens compiladas tradicionais, mas, no Python, o compilador está sempre presente em tempo de execução e faz parte do sistema que executa programas.

Isso propicia um ciclo de desenvolvimento muito mais rápido – não há necessidade de compilação e ligação prévia antes que a execução possa começar; basta digitar e executar o código. Isso também acrescenta uma característica muito mais dinâmica à linguagem – é possível e, freqüentemente, muito conveniente, que os programas em Python construam e executem outros programas em Python em tempo de execução. As rotinas internas `eval` e `exec`, por

exemplo, aceitam e executam strings contendo código de programa em Python. Essa estrutura presta-se para a personalização do produto – como o código em Python pode ser alterado dinamicamente, os usuários podem modificar as partes que estão em Python de um sistema no local, sem a necessidade de ter ou de compilar o código de sistema inteiro.

VARIAÇÕES DO MODELO DE EXECUÇÃO

Antes de prosseguirmos, devemos mencionar que o fluxo de execução interno, descrito na seção anterior, reflete a implementação padrão atual do Python e não é realmente um requisito da linguagem em si. Por isso, o modelo de execução está propenso a mudar com o tempo. Na verdade, já existem alguns sistemas que modificam um pouco o quadro da Figura 2-2. Vamos explorar brevemente as variações mais importantes.

Alternativas de implementação do Python

Na realidade, quando este livro estava sendo escrito, havia duas implementações principais da linguagem Python – *CPython* e *Jython* –, junto com diversas implementações secundárias, como a *Python.NET*. O CPython é a implementação padrão; todas as outras têm propósitos e funções muito específicos. Todas implementam a mesma linguagem Python, mas executam os programas de forma diferente.

CPython

A implementação original e padrão do Python é normalmente chamada de CPython, quando você quer compará-la com as outras duas. Seu nome decorre do fato de que ela é desenvolvida em código de linguagem ANSI C portável. Esse é o Python que você busca no endereço *www.python.org*, recebe com a distribuição ActivePython e tem automaticamente na maioria das máquinas Linux. Se você tiver encontrado uma versão previamente instalada do Python em sua máquina, ela provavelmente também é CPython, a não ser que sua empresa esteja usando o Python de maneira muito especializada.

A menos que você queira desenvolver script de aplicativos em Java ou .NET, provavelmente desejará usar o sistema CPython padrão. Como essa é a implementação de referência da linguagem, ela tende a executar com mais velocidade, ser mais completa e mais robusta do que os sistemas alternativos. A Figura 2-2 reflete a arquitetura de tempo de execução do CPython.

Jython

O sistema Jython (originalmente conhecido como JPython) é uma implementação alternativa da linguagem Python, destinada à integração com a linguagem de programação Java. O Jython consiste em classes Java que compilam código-fonte Python em código de byte Java e, depois, direcionam o código de byte resultante para a Máquina Virtual Java (JVM). Normalmente, os programadores ainda escrevem instruções Python em arquivos de texto *.py*; basicamente, o sistema Jython apenas substitui as duas bolhas da direita na Figura 2-2 por equivalentes baseados em Java.

O objetivo do Jython é permitir que o código Python resulte em script de aplicativos Java, de forma muito parecida como o CPython resulta em script de componentes C e C++. Sua integração com Java é notadamente transparente. Como o código Python é transformado em código de byte Java, em tempo de execução ele parece e se comporta como um verdadeiro

programa em Java. Os scripts Jython podem servir como applets e servlets da Web, como GUIs baseadas em Java etc. Além disso, o Jython inclui suporte de integração que permite ao código Python importar e usar classes Java como se fossem codificadas em Python. Como o Jython é mais lento e menos robusto do que o CPython, ele é visto, normalmente, como uma ferramenta de interesse principalmente para desenvolvedores de Java.

Python.NET

Uma terceira (e ainda experimental) implementação do Python é projetada para permitir que os programas Python integrem-se com aplicativos desenvolvidos para a estrutura .NET da Microsoft. A estrutura .NET e seu sistema de tempo de execução em linguagem de programação C# são projetados para serem uma camada de comunicação de objetos neutra quanto à linguagem, no espírito do modelo COM mais antigo da Microsoft. O Python.NET permite que os programas em Python atuem como componentes clientes e servidores, acessíveis a partir de outras linguagens .NET.

Por implementação, o Python.NET é muito mais parecido com o Jython – ele substitui as duas últimas bolhas da Figura 2-2 por execução no ambiente .NET. Também como o Jython, o Python.NET tem um foco especial – ele é interessante principalmente para desenvolvedores que estejam integrando o Python com componentes .NET. (A evolução do Python.NET não estava clara quando escrevemos isto; para obter mais detalhes, consulte os recursos on-line do Python.)

O compilador just-in-time Psyco

O CPython, o Jython e o Python.NET implementam a linguagem Python de maneira semelhante: compilando o código-fonte em código de byte e executando o código de byte em uma máquina virtual apropriada. O sistema *Psyco* não é outra implementação do Python, mas um componente que estende o modelo de execução de código de byte para fazer os programas serem executados mais rapidamente. Em termos da Figura 2-2, o Psyco é um aprimoramento da PVM que reúne e utiliza informações de tipo enquanto o programa executa, para transformar partes do código de byte do programa em código de máquina binário, a fim de obter uma execução mais rápida. O Psyco realiza a transformação sem exigir alterações no código ou uma etapa de compilação separada, durante o desenvolvimento.

Grosso modo, enquanto seu programa executa, o Psyco reúne informações sobre os tipos de objetos e, então, substitui a parte correspondente do código de byte por uma equivalente em código de máquina, para acelerar a execução global do seu programa. O resultado é que, com o Psyco, seu programa se torna muito mais rápido com o passar do tempo e quando está em execução. Em casos ideais, alguns código em Python, compilados sob o Psyco, podem se tornar tão rápidos quanto um código em C.

Como essa transformação de código de byte acontece em tempo de execução do programa, o Psyco é geralmente conhecido como um compilador *just-in-time* (JIT). Na verdade, o Psyco é um pouco mais do que os compiladores JIT que você possa ter visto para a linguagem Java. Na realidade, o Psyco é um *compilador JIT especializado* – ele gera código de máquina personalizado para os tipos de dados que seu programa realmente usa. Por exemplo, se uma parte do seu programa usa tipos de dados diferentes em diferentes momentos, o Psyco pode gerar uma versão diferente de código de máquina para suportar cada combinação de tipo diferente.

O Psyco tem mostrado que acelera código em Python substancialmente. De acordo com sua página na Web, o Psyco fornece "aumentos de velocidade de `2x` a `100x`, normalmente `4x`, com

um interpretador Python não modificado e código-fonte não modificado e apenas um módulo de extensão C carregável dinamicamente". Igualmente importante, os maiores aumentos de velocidade são percebidos em código de algoritmo escrito em Python puro – exatamente os tipos de código que você normalmente poderia migrar para a linguagem C, para otimizar. Com o Psyco, tais migrações se tornam ainda menos importantes.

O Psyco ainda não é uma parte padrão do Python; você terá que procurá-lo e instalá-lo separadamente. Ele também é um projeto de pesquisa; portanto, você terá que acompanhar sua evolução on-line. Para obter mais detalhes sobre a extensão Psyco e outros trabalhos de JIT que podem surgir, consulte o endereço *http://www.python.org*; a home page do Psyco atualmente se encontra no endereço *http://psyco.sourceforge.net*.

Binários congelados

Às vezes, quando as pessoas pedem um compilador de Python "real", o que elas estão realmente procurando é simplesmente uma maneira de gerar um executável binário independente, a partir de seus programas em Python. Isso é mais uma idéia de empacotamento e distribuição do que um conceito de fluxo de execução, mas tem alguma relação. Com a ajuda de ferramentas de outros fornecedores, que você pode buscar na Web, é possível transformar seus programas em Python em verdadeiros executáveis – conhecidos no mundo Python como *binários congelados*.

Os binários congelados empacotam o código de byte junto com a PVM (interpretador) e todos os arquivos de suporte Python que seu programa precisa, em um único pacote. Existem algumas variações sobre esse tema, mas o resultado final pode ser um único programa binário executável (por exemplo, um arquivo *.exe* no Windows), que pode ser facilmente distribuído para os clientes. Na Figura 2-2, é como se o código de byte e a PVM fossem fundidos em um único componente – um arquivo binário congelado.

Atualmente, três sistemas principais são capazes de gerar binários congelados: o *Py2exe* (para Windows), o *Installer* (semelhante, mas também funciona em Linux e Unix, e é capaz de gerar binários de instalação automática) e o *Freeze* (o original). Talvez você tenha que procurar essas ferramentas separadamente do Python, mas elas estão disponíveis gratuitamente. Elas também estão constantemente evoluindo; portanto, consulte o endereço *http://www.python.org* e o site da Web Vaults of Parnassus para obter mais informações sobre essas ferramentas. Para dar uma idéia da abrangência desses sistemas, o Py2exe pode congelar programas independentes que usam as bibliotecas de GUI Tkinter, Pmw, wxPython e PyGTK; programas que usam o kit de ferramentas de programação de jogos *pygame*; programas clientes win-32com; e muito mais.

Os binários congelados não são o mesmo que a saída de um verdadeiro compilador – eles executam código de byte por meio de uma máquina virtual. Assim, fora uma possível melhoria na inicialização, os binários congelados são executados na mesma velocidade dos arquivos-fonte originais. Eles também não são pequenos (pois contêm a PVM), mas não são demasiadamente grandes pelos padrões de tamanho atuais. Como o Python é incorporado no binário congelado, não precisa ser instalado no usuário final para executar seu programa. Além disso, como seu código é incorporado no binário congelado, ele fica efetivamente oculto para o usuário final.

Esse esquema de empacotamento em um único arquivo é particularmente atraente para desenvolvedores de software comercial. Por exemplo, um programa de interface com o usuário baseado no kit de ferramentas Tkinter pode ser congelado em um arquivo executável e distri-

buído como um programa auto-suficiente no CD ou na Web. Os usuários finais não precisam instalar e nem mesmo conhecer o Python.

Possibilidades futuras?

Finalmente, observe que o modelo de execução de runtime esboçado aqui é na realidade um artefato da implementação corrente e não a linguagem em si. Por exemplo, não é impossível que um compilador tradicional completo, de código-fonte em Python para código de máquina, possa aparecer durante o tempo em que este livro permanecer nas prateleiras (embora não tenha aparecido nenhum há mais de uma década). Novos formatos de código de byte e novas variantes de implementação também podem ser adotados no futuro. Por exemplo:

- O emergente projeto *Parrot* pretende fornecer um formato de código de byte, máquina virtual e técnicas de otimização comuns para uma variedade de linguagens de programação (veja *http://www.python.org*).
- O sistema *Stackless Python* é uma variante da implementação de CPython padrão que não salva o estado na pilha de chamadas da linguagem C. Isso torna o Python mais facilmente adaptado para pequenas arquiteturas de pilha e abre novas possibilidades de programação, como as co-rotinas.
- O novo projeto *PyPy* é uma tentativa de reimplementar a PVM no próprio Python, para permitir novas técnicas de implementação.

Embora tais esquemas de implementação futuros possam alterar um pouco a estrutura de tempo de execução do Python, parece provável que o compilador de código de byte ainda será o padrão por algum tempo. A portabilidade e a flexibilidade em tempo de execução do código de byte são características importantes para muitos sistemas Python. Além disso, adicionar declarações de restrição de tipo, para suportar compilação estática, prejudicaria a flexibilidade, a compacidade, a simplicidade e o espírito geral do desenvolvimento em Python. Devido à natureza altamente dinâmica do Python, qualquer implementação futura provavelmente manterá muitos artefatos da PVM atual.

3

Como Você Executa Programas

Certo, é hora de começar a executar algum código. Agora que você tem uma idéia da execução de programas, finalmente está pronto para começar a fazer alguma programação real em Python. Neste ponto, vamos supor que você já tenha instalado o Python em seu computador; se não tiver, consulte o Apêndice A para ver dicas de instalação e configuração.

Existem várias maneiras de fazer o Python executar o código que você digita. Este capítulo discute todas as técnicas de execução de programas em uso comum atualmente. Nesse meiotempo, você aprenderá a digitar o código *interativamente*, a salvá-lo em *arquivos* para ser executado com linhas de comando, truques do Unix, cliques em ícones, IDEs, importações e muito mais.

Se você quer apenas descobrir como faz para executar um programa em Python rapidamente, talvez sinta a tentação de ler somente as partes pertinentes à sua plataforma e passar para o Capítulo 4. Mas não pule o material sobre importações de módulo, pois é fundamental para entender a arquitetura do Python. Além disso, o convidamos a pelo menos ler superficialmente as seções sobre IDLE e outros IDEs, para que você saiba quais ferramentas estão disponíveis quando começar a desenvolver programas mais sofisticados em Python.

DESENVOLVIMENTO INTERATIVO

Talvez a maneira mais simples de executar programas Python seja digitá-los na linha de comando interativa da linguagem. Existem diversas maneiras de iniciar essa linha de comando – em um IDE, a partir do console do sistema etc. Supondo que o interpretador esteja instalado como um programa executável em seu sistema, a maneira mais neutra quanto à plataforma de iniciar uma sessão do interpretador interativo normalmente é digitar apenas “python” no prompt do seu sistema operacional, sem nenhum argumento. Por exemplo:

```
% python
Python 2.2 (#28, Dec 21 2001, 12:21:22) [MSC 32 bits (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Aqui, a palavra "python" é digitada no prompt de shell do seu sistema para iniciar uma sessão interativa do Python (o caractere "%" representa o prompt do seu sistema e não uma entrada sua). A noção de *prompt de shell do sistema* é genérica, mas varia de acordo com a plataforma:

- No Windows, você pode digitar **python** em uma janela de console do DOS (também conhecida como Prompt de comando) ou na caixa de diálogo Iniciar/Executar.
- No Unix e no Linux, você pode digitar isso em uma janela de shell (por exemplo, em um *xterm* ou console, executando um shell como *ksh* ou *csh*).
- Outros sistemas podem usar dispositivos semelhantes ou específicos da plataforma. No PalmPilot, por exemplo, dê um clique no ícone de base do Python para ativar uma sessão interativa; em um PDA Zaurus, abra uma janela Terminal.

Se você não configurou a variável de ambiente PATH do seu shell para incluir o Python, talvez precise substituir a palavra "python" pelo caminho completo para o executável Python em sua máquina. Por exemplo, no Windows, tente digitar **C:\Python22\python** (ou **C:\Python23\python**, para a versão 2.3); no Unix e no Linux, **/usr/local/bin/python** (ou **/usr/bin/python**), freqüentemente será suficiente.

Uma vez iniciada a sessão interativa do Python, ela começa imprimindo duas linhas de texto informativo (que normalmente omitimos em nossos exemplos para economizar espaço) e solicita entrada com >>>, quando está esperando que você digite uma nova instrução ou expressão da linguagem. Ao se trabalhar interativamente, os resultados do seu código são exibidos após as linhas >>> – aqui estão os resultados de duas instruções print do Python:

```
% python
>>> print 'Hello world!'
Hello world!
>>> print 2 ** 8
256
```

Novamente, não se preocupe ainda com os detalhes das instruções `print` mostradas aqui (começaremos a nos aprofundar na sintaxe, no próximo capítulo). Em resumo, elas imprimem uma string e um inteiro do Python, conforme mostrado pelas linhas de saída que aparecem após cada linha de entrada >>>.

Ao trabalharmos interativamente dessa forma, podemos digitar quantos comandos Python quisermos; cada um é executado imediatamente após ser digitado. Além disso, como a sessão interativa imprime automaticamente os resultados das expressões digitadas, normalmente não precisamos escrever "print" explicitamente nesse prompt:

```
>>> lumberjack = 'okay'
>>> lumberjack
'okay'
>>> 2 ** 8
256
>>>                          use Ctrl-D ou Ctrl-Z para sair
%
```

Aqui, as duas últimas linhas digitadas são expressões (`lumberjack e 2 ** 8`) e seus resultados são exibidos automaticamente. Para sair de uma sessão interativa como essa e voltar para o prompt de shell do seu sistema, digite Ctrl-D em máquinas do tipo Unix; nos sistemas MS-DOS e Windows, digite Ctrl-Z. Na GUI IDLE, discutida posteriormente, digite Ctrl-D ou simplesmente feche a janela.

Agora, não estamos fazendo muita coisa no código dessa sessão: digitamos as instruções `print` e de atribuição do Python e algumas expressões, as quais estudaremos em detalhes posteriormente. O principal a notar é que o código digitado é executado imediatamente pelo interpretador, quando a tecla Enter é pressionada no fim da linha.

Por exemplo, após digitar a primeira instrução `print` no prompt `>>>`, a saída (uma string do Python) é ecoada imediatamente. Não há necessidade de primeiro passar o código por um compilador e por um linkeditor, como você normalmente faria ao utilizar uma linguagem como C ou C++. Conforme você verá em capítulos posteriores, também é possível executar instruções de várias linhas no prompt interativo; a instrução é executada imediatamente após você ter digitado todas as suas linhas.

Além de digitar **python** em uma janela de shell, você também pode começar sessões interativas semelhantes iniciando a janela principal do IDLE ou no Windows por meio dos menus do Python do botão Iniciar e selecionando a opção de menu (linha de comando), como se vê na Figura 2-1. As duas maneiras geram um prompt `>>>` com funcionalidade equivalente – o código é executado assim que é digitado.

Testando o código no prompt interativo

Como o código é executado imediatamente, o prompt interativo revela-se um excelente lugar para experimentar a linguagem. Ele será usado freqüentemente neste livro para demonstrar exemplos menores. Na verdade, esta é a primeira regra básica a ser lembrada: se você estiver em dúvida a respeito de como um trecho de código Python funciona, ative a linha de comando interativa e experimente para ver o que acontece. São boas as chances de que você não estrague nada. (Você precisa saber mais sobre interfaces de sistema antes de se tornar perigoso).

Embora você não faça a maior parte do seu desenvolvimento em sessões interativas (porque o código digitado lá não é salvo), o interpretador interativo é um lugar excelente para testar o código colocado em arquivos. Você pode importar seus arquivos de módulo interativamente e executar testes nas ferramentas que eles definem, digitando chamadas no prompt interativo. Na maioria das vezes, o prompt interativo é um lugar para testar componentes de programa, independente de sua fonte – você pode digitar chamadas para funções C vinculadas, testar classes Java no Jython e muito mais. Parcialmente devido a sua natureza interativa, o Python suporta um estilo de programação experimental e exploratório que você achará conveniente quando começar a usar a linguagem.

Usando o prompt interativo

Embora o prompt interativo seja simples de usar, existem algumas maneiras pelas quais ele parece atrapalhar os iniciantes:

Digite apenas comandos Python. Antes de tudo, lembre-se de que você só pode digitar código Python no prompt do Python, e não comandos de sistema. Existem maneiras de executar comandos de sistema dentro do código Python (por exemplo, `os.system`), mas elas não são tão diretas como simplesmente digitar o comando em si.

As instruções print são exigidas apenas em arquivos. Como o interpretador interativo imprime automaticamente os resultados das expressões, você não precisa digitar interativamente as instruções `print` completas. Essa é uma característica interessante, mas tende a confundir os usuários quando eles passam a escrever código em arquivos. Dentro de um arquivo de código, você deve usar as instruções `print` para ver a saída, pois os resultados

das expressões não repercutem automaticamente. Você deve escrever print nos arquivos, mas não interativamente.

Não faça endentação no prompt interativo (ainda). Ao digitar programas em Python, interativamente ou em um arquivo de texto, certifique-se de começar na coluna 1 (isto é, totalmente à esquerda) todas as suas instruções não testadas. Se você não fizer isso, o Python poderá imprimir uma mensagem "SyntaxError". Até o Capítulo 9, nenhuma instrução será testada; portanto, isso inclui tudo, por enquanto. Essa parece ser uma confusão recorrente nas aulas introdutórias sobre Python. Um espaço no início gera uma mensagem de erro.

Prompts e instruções compostas. Não veremos instruções compostas (de várias linhas) até o Capítulo 9, mas, como uma prévia, você deve saber que ao digitar interativamente as linhas 2 e acima de uma instrução composta, o prompt poderá mudar. Na interface de janela de shell simples, o prompt interativo muda para..., em vez de >>>, para as linhas 2 e acima; na interface IDLE, as linhas após a primeira são endentadas automaticamente. Em qualquer caso, uma linha em branco (pressionar a tecla Enter no início de uma linha) é necessária para informar ao Python interativo que você terminou de digitar uma instrução de várias linhas; em contraste, as linhas em branco são ignoradas em arquivos.

Você vai ver porque isso importa, no Capítulo 9. Por enquanto, se acontecer de encontrar um prompt... ou uma linha em branco, quando digitar seu código, isso provavelmente significa que, de algum modo, você confundiu o Python interativo, fazendo-o pensar que estava digitando uma instrução de várias linhas. Tente pressionar a tecla Enter ou a combinação Ctrl-C para voltar ao prompt principal. Os prompts >>> e... também podem ser alterados (eles estão disponíveis no módulo interno `sys`), mas vamos supor que não tenham sido, em nossos exemplos.

LINHAS DE COMANDO DE SISTEMA E ARQUIVOS

Embora o prompt interativo seja excelente para fazer experiências e testes, ele tem uma enorme desvantagem: os programas nele digitados desaparecem assim que o interpretador do Python os executa. O código que você digita interativamente nunca é armazenado em um arquivo; portanto, você não pode executá-lo novamente sem digitá-lo outra vez, desde o início. Operações de recorte e colagem e de recall de comando podem ajudar um pouco aqui, mas não muito, especialmente quando você começa a escrever programas maiores. Para recortar e colar o código de uma sessão interativa, você precisa editar prompts do Python, saídas de programa etc.

Para salvar programas permanentemente, você precisa gravar seu código em arquivos, normalmente conhecidos como *módulos*. Os módulos são simplesmente arquivos de texto que contêm instruções Python. Uma vez escritos, você pode pedir ao interpretador do Python para que execute as instruções em tais arquivos qualquer número de vezes e de diversas maneiras – por meio de linhas de comando do sistema, por meio de cliques em ícones de arquivo, por meio de opções na interface com o usuário IDLE e muito mais. Independente de como eles sejam executados, o Python executa, do início ao fim, todo o código de um arquivo de módulo, sempre que você executa o arquivo. Esses arquivos são freqüentemente referidos como *programas* no Python – uma série de instruções previamente desenvolvidas.

As próximas seções exploram maneiras de executar o código digitado em arquivos de módulo. Nesta seção, executamos arquivos da maneira mais básica: listando seus nomes em uma linha de comando Python digitada no prompt do sistema. Como primeiro exemplo, suponha que iniciemos nosso editor de textos predileto (por exemplo, o vi, o notepad ou o editor IDLE) e digitemos duas instruções Python em um arquivo de texto chamado *spam.py*:

```
print 2 ** 8                              # Eleva a uma potência
print 'the bright side ' + 'of life'      # + significa concatenação
```

Esse arquivo contém duas instruções `print` e *comentários* do Python à direita. O texto após # é simplesmente ignorado (sendo considerado como comentário legível para seres humanos) e não faz parte da sintaxe da instrução. Novamente, ignore a sintaxe do código nesse arquivo, por enquanto. O ponto a ser notado é que digitamos o código em um arquivo, em vez do prompt interativo. Nesse processo, desenvolvemos um script Python totalmente funcional.

Uma vez que tivermos salvo esse arquivo de texto, poderemos pedir ao Python para que o execute, listando seu nome de arquivo completo como primeiro argumento em um comando Python, digitado no prompt de shell do sistema:

```
% python spam.py
256
the bright side of life
```

Aqui, novamente, você digitará o comando de shell no que seu sistema fornecer para entrada de linha de comando – uma janela de console do DOS, um xterm ou algo semelhante. Lembre-se de substituir "python" por um caminho de diretório completo, caso sua variável de ambiente PATH não esteja configurada. A saída desse pequeno script aparece após o comando ser digitado – é o resultado das duas instruções print presentes no arquivo de texto.

Note que o arquivo de módulo é chamado *spam.py*. No tocante a todos os arquivos de nível superior, ele poderia ser chamado simplesmente *spam*, mas os arquivos de código que queremos importar para um cliente precisam terminar com o sufixo *.py*. Estudaremos as importações posteriormente neste capítulo. Como talvez você queira importar um arquivo no futuro, é uma boa idéia utilizar sufixos *.py* para a maioria dos arquivos Python que escrever. Alguns editores de textos detectam os arquivos Python pelo sufixo *.py*; se o sufixo não estiver presente, você poderá não obter coisas como cores em sintaxe e endentação automática.

Como esse esquema usa linhas de comando de shell para iniciar programas em Python, toda a sintaxe de shell normal se aplica. Por exemplo, podemos direcionar a saída de um script Python para um arquivo, para salvá-lo, usando a sintaxe de shell especial:

```
% python spam.py > saveit.txt
```

Nesse caso, as duas linhas de saída mostradas na execução anterior aparecem no arquivo *saveit.txt*, em vez de serem impressas. Geralmente, isso é conhecido como *redirecionamento de fluxo*; funciona para texto de entrada e de saída, tanto em sistemas Windows como do tipo Unix. Isso também tem pouco a ver com o Python (o Python simplesmente suporta o redirecionamento); portanto, vamos pular os detalhes sobre redirecionamento aqui.

Esteja você trabalhando em uma plataforma Windows ou MS-DOS, este exemplo funciona de forma igual, mas o prompt de sistema normalmente é diferente:

```
C:\Python22>python spam.py
256
the bright side of life
```

Como sempre, certifique-se de digitar o caminho completo para o Python, caso não tenha configurado sua variável de ambiente PATH:

```
D:\temp>C:\python22\python spam.py
256
the bright side of life
```

(Em algumas versões do Windows, você também pode digitar apenas o nome do seu script, independente do diretório em que estiver trabalhando. Como os sistemas Windows mais recentes utilizam o registro do Windows para encontrar um programa para executar um arquivo, você não precisa listá-lo explicitamente na linha de comando.

Finalmente, lembre-se de fornecer o caminho completo para seu arquivo de script, caso ele esteja em um diretório diferente daquele em que está trabalhando. Por exemplo, a linha de comando de sistema a seguir, executada a partir de *D:\other*, presume que o Python está em seu caminho de sistema, mas executa um arquivo localizado em outro lugar:

```
D:\other>python c:\code\mystrip.py
```

Usando linhas de comando e arquivos

Executar arquivos de programa a partir de linhas de comando também é uma opção de execução muito simples, especialmente se você estiver familiarizado com as linhas de comando em geral, de algum trabalho anterior com Linux ou DOS. Aqui estão algumas indicações sobre armadilhas comuns para iniciantes, antes de prosseguirmos:

Cuidado com as extensões automáticas no Windows. Se você usa o programa Notepad para escrever arquivos de programa no Windows, tome o cuidado de selecionar todos os arquivos quando chegar a hora de salvar seu arquivo, e forneça para seu arquivo o sufixo. *py* explicitamente. Caso contrário, o Notepad salvará o arquivo com a extensão ".txt" (por exemplo, como `spam.py.txt`), tornando-o difícil de executar em alguns esquemas de execução.

Pior ainda, o Windows oculta as extensões de arquivo por padrão, a não ser que você tenha alterado suas opções de visualização; portanto, você pode nem notar que escreveu um arquivo de texto e não um arquivo do Python. O ícone do arquivo pode revelar isso – se não for uma cobra, você pode ter problemas. Código sem cores no IDLE e arquivos que se abrem para edição, em vez de executarem quando recebem um clique, são outros sintomas desse problema.

De modo semelhante, o MS Word adiciona a extensão *.doc* por padrão; muito pior ainda, ele adiciona caracteres de formatação que não são válidos na sintaxe do Python. Como regra básica, sempre selecione todos os arquivos ao salvar no Windows ou use editores de texto mais amigáveis para o programador, como o IDLE. O IDLE nem mesmo adiciona o sufixo *.py* automaticamente – um recurso que os programadores gostam e os usuários não.

Use extensões de arquivo em prompts de sistema, mas não em importações. Não se esqueça de digitar o nome completo do seu arquivo em linhas de comando de sistema, ou seja, use python `spam.py`. Isso é diferente das instruções de importação do Python que veremos posteriormente neste capítulo, as quais omitem o sufixo de arquivo *.py* e o caminho de diretório, como em `import spam`. Isso pode parecer simples, mas é um erro comum.

No prompt do sistema, você está em um shell de sistema e não no Python; portanto, as regras de pesquisa de arquivo de módulo do Python não se aplicam. Por isso, você deve fornecer a extensão *.py* e, opcionalmente, pode incluir o caminho de diretório completo que leva ao arquivo que deseja executar. Por exemplo, para executar um arquivo residente em um diretório diferente daquele em que está trabalhando, você normalmente lista o nome do caminho completo (`C:\python22>python d:\tests\spam.py`). Dentro do código

Python, você escreve apenas `import spam` e conta com o caminho de pesquisa de módulo da linguagem para localizar seu arquivo.

Use instruções print em arquivos. Sim, isso já foi mencionado na seção anterior, mas é tão importante que deve ser dito novamente aqui. Ao contrário do desenvolvimento interativo, você geralmente deve usar instruções `print` para ver a saída de arquivos de programa.

Scripts executáveis Unix (#!)

Se você for usar o Python em um sistema Unix, Linux ou qualquer um do tipo Unix, também pode transformar arquivos de código Python em programas executáveis, como faria para programas desenvolvidos em uma linguagem de shell, como *csh* ou *ksh*. Tais arquivos normalmente são chamados de scripts executáveis. Em termos simples, os scripts executáveis do estilo Unix são apenas arquivos de texto normais contendo instruções Python, mas com duas propriedades especiais:

A primeira linha é especial. Normalmente, os scripts começam com uma primeira linha iniciada com os caracteres #! (freqüentemente chamados de "hash bang"), seguidos do caminho até o interpretador do Python em sua máquina.

Normalmente, eles têm privilégios de executáveis. Os arquivos de script normalmente são marcados como executáveis para informar o sistema operacional que eles podem ser executados como programas de nível superior. Nos sistemas Unix, um comando como `chmod +x file.py` normalmente resolve.

Vamos ver um exemplo. Suponha que usemos um editor de textos novamente, para criar um arquivo de código Python chamado `brian`:

```
#!/usr/local/bin/python
print 'The Bright Side of Life...'              # Outro comentário aqui
```

A linha especial no início do arquivo informa ao sistema onde está o interpretador do Python. Tecnicamente, a primeira linha é um comentário do Python. Conforme mencionado anteriormente, todos os comentários nos programas em Python começam com `#` e se estendem até o final da linha; eles são lugares para inserir informações extras para leitores humanos de seu código. Mas quando aparece um comentário como a primeira linha desse arquivo, ele é especial, pois o sistema operacional o utiliza para localizar um interpretador para executar o código do programa presente no restante do arquivo.

Além disso, esse arquivo é chamado simplesmente `brian`, sem o sufixo *.py* usado anteriormente para o arquivo de módulo. Adicionar *.py* no nome não prejudicaria (e poderia nos ajudar a lembrar que esse é um arquivo de programa do Python), mas como não pretendemos deixar que outros módulos importem o código desse arquivo, o nome do arquivo é irrelevante. Se dermos ao arquivo privilégios de executável com o comando de shell `chmod +x brian`, poderemos executá-lo a partir do shell do sistema operacional, como se ele fosse um programa binário:

```
% brian
The Bright Side of Life...
```

Uma nota para usuários do Windows: o método descrito aqui é um truque do Unix e pode não funcionar em sua plataforma. Não se preocupe. Basta usar a técnica de linha de comando

básica, explorada anteriormente. Liste o nome do arquivo em uma linha de comando explícita do Python:*

```
C:\book\tests> python brian
The Bright Side of Life...
```

Neste caso, você não precisa do comentário #! especial no início (embora o Python apenas o ignore, se estiver presente) e o arquivo não precisa receber privilégios de executável. Na verdade, se você quiser executar arquivos de forma portável entre o Unix e o MS Windows, sua vida provavelmente será mais simples se sempre usar a estratégia de linha de comando básica para executar programas e não scripts de estilo Unix.

CLICANDO EM ÍCONES DE ARQUIVO DO WINDOWS

No Windows, o Python registra-se automaticamente como o programa que abre arquivos da linguagem quando recebe um clique. Por isso, é possível executar os programas em Python que você escreve apenas dando um clique (ou dando um clique duplo) em seus ícones de arquivo com o mouse.

No Windows, os cliques em ícones se tornam fáceis por causa do registro. Em sistemas que não são Windows, você provavelmente poderá realizar um truque semelhante, mas os ícones, o explorador de arquivos, os esquemas de navegação e outros, podem ser ligeiramente diferentes. Em alguns sistemas Unix, por exemplo, talvez você precise registrar a extensão *.py* na GUI de seu explorador de arquivos, tornar seu script executável usando o truque dos caracteres #! da seção anterior ou associar o tipo de arquivo MIME a um aplicativo ou comando, editando arquivos, instalando programas ou usando outras ferramentas. Consulte a documentação de seu explorador de arquivos para obter mais detalhes, caso os cliques não funcionem corretamente na primeira tentativa.

Clicando em ícones no Windows

Para ilustrar, suponha que criemos o seguinte arquivo de programa com nosso editor de textos e o salvemos com o nome de arquivo *script4.py*:

```
# Um comentário
import sys
print sys.platform
print 2 ** 100
```

Não há muita novidade aqui – apenas uma instrução `import` e, novamente, duas instruções `print` (`sys.platform` é apenas uma string que identifica o tipo de computador em que você está trabalhando; ela fica em um módulo chamado `sys`, que devemos importar para carregar). Na verdade, podemos executar esse arquivo a partir da linha de comando do sistema:

```
D:\OldVaio\LP-2ndEd\Examples>c:\python22\python script4.py
win32
1267650600228229401496703205376
```

* Conforme foi visto ao explorarmos as linhas de comando, as versões modernas do Windows também permitem que você digite apenas o nome de um arquivo `.py` na linha de comando do sistema – elas usam o registro para saber como abrir o arquivo com o Python (por exemplo, digitar `brian.py` é equivalente a digitar `python brian.py`). Esse modo de linha de comando tem um espírito semelhante aos caracteres #! do Unix. Note que alguns *programas* podem realmente interpretar e usar uma primeira linha `#!` no Windows, de modo muito parecido com o Unix, mas o shell do sistema DOS e o próprio Windows a ignoram completamente.

O truque da pesquisa env do Unix

Em alguns sistemas Unix, você pode evitar escrever do caminho até o interpretador do Python, escrevendo o comentário da primeira linha especial, como segue:

```
#!/usr/bin/env python
... o script fica aqui ...
```

Quando escrito dessa maneira, o programa `env` localiza o interpretador do Python, de acordo com as configurações do caminho de pesquisa do seu sistema (isto é, na maioria dos shells Unix, pesquisando em todos os diretórios listados na variável de ambiente PATH). Esse esquema pode ser mais portável, pois você não precisa escrever um caminho de instalação do Python na primeira linha de todos os seus scripts.

Contanto que você tenha acesso ao programa `env` em todos os lugares, seus scripts serão executados independentemente de onde o Python estiver em seu sistema – você só precisa alterar as configurações da variável de ambiente PATH entre plataformas e não a primeira linha de todos os seus scripts. É claro que isso presume que o programa `env` fica no mesmo local em todos os lugares (em algumas máquinas, ele também pode estar em */sbin*, */bin* ou outro); senão, todas as possibilidades de portabilidade serão perdidas.

Os cliques no ícone nos permitem executar esse arquivo sem nenhuma digitação. Se encontrarmos o ícone desse arquivo – por exemplo, selecionando Meu Computador e indo até a unidade de disco D –, obteremos a imagem do explorador de arquivos capturada na Figura 3-1 e apresentada no Windows XP. Os arquivos-fonte do Python aparecem como cobras no Windows e os arquivos de código de byte aparecem como cobras de olhos fechados (ou com uma cor avermelhada, na versão 2.3). Normalmente, você desejará dar um clique (ou executar de outra forma) no arquivo de código-fonte para por em ordem suas alterações mais recentes. Para executar o arquivo aqui, basta dar um clique no ícone de *script4.py*.

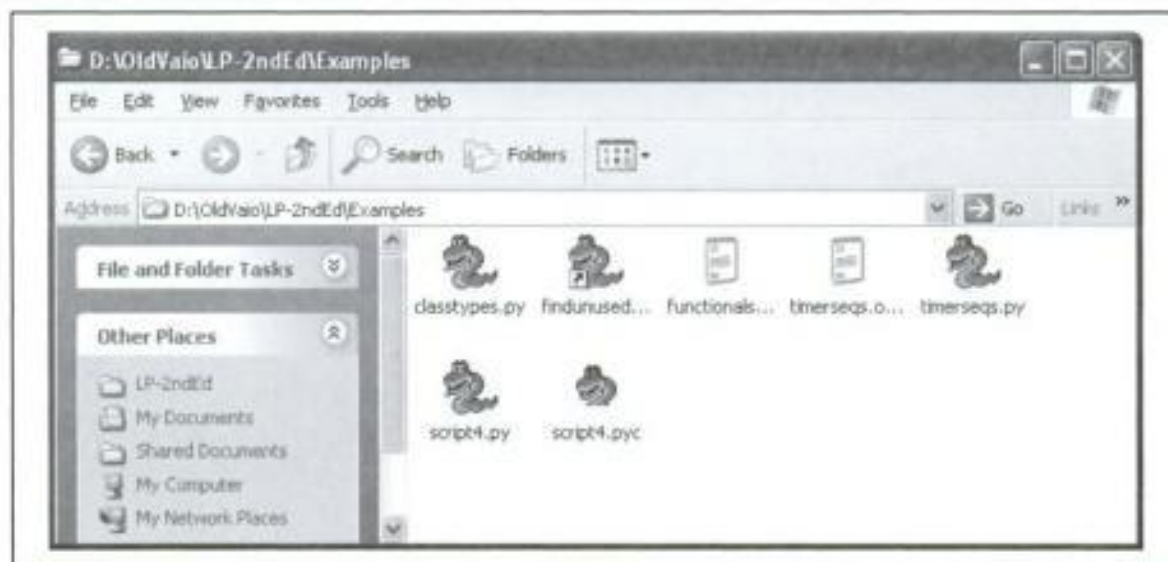



Figura 3-1 Ícones de arquivo do Python no Windows.

O truque raw_input

Infelizmente, no Windows, o resultado de clicar em ícones de arquivo pode não ser totalmente satisfatório. Na verdade, este script de exemplo gera um "clarão" desconcertante, quando recebe um clique – não é o tipo de retorno que os programadores de Python normalmente

esperam! Não se trata de um erro, mas está relacionado com a maneira pela qual a porta do Windows manipula a saída impressa.

Por padrão, o Python gera uma janela instantânea de console DOS na cor preta, para servir como entrada e saída de um arquivo que recebeu um clique. Se um script imprime e sai, então, bem, ele apenas imprime e sai – a janela do console aparece e texto é impresso nela, mas ela se fecha e desaparece na saída do programa. A não ser que você seja muito rápido ou que sua máquina seja muito lenta, a saída não poderá ser vista. Embora esse seja um comportamento normal, provavelmente não é o que você tinha em mente.

Felizmente, é fácil resolver isso. Se você precisa que a saída do seu script fique visível mais um pouco de tempo quando executado com cliques, basta colocar uma chamada para a função interna `raw_input` no final do seu script. Por exemplo:

```
# Um comentário
import sys
print sys.platform
print 2 ** 100
raw_input()                          # ADICIONADO
```

Em geral, `raw_input` lê a próxima linha da entrada padrão e espera, caso ainda não haja nenhuma linha disponível. O resultado, neste contexto, será uma pausa no script, mantendo com isso a janela de saída aberta (mostrada na Figura 3-2), até que pressionemos a tecla Enter.

Figura 3-2 Saída de programa que recebeu um clique com raw_input.

Agora que já mostramos esse truque, lembre-se de que ele só é exigido para o Windows, apenas se seu script imprime texto e sai, e somente se você ativar esse script dando um clique no ícone do seu arquivo. Você só deve adicionar essa chamada no final dos seus arquivos de nível superior, se, e somente se, todas essas três condições se aplicarem. Não há motivo para adicionar essa chamada em nenhum outro contexto.*

Antes de prosseguirmos, note que a chamada de `raw_input` aplicada aqui é a correlata para entrada do uso da instrução `print` para saídas. Ela é a maneira mais simples de ler entrada de usuário e é mais geral do que este exemplo implica. Por exemplo, `raw_input`:

- Opcionalmente, aceita uma string que será impressa como um prompt (por exemplo, `raw_input('Press Enter to exit')`)
- Retorna para seu script a linha de texto lida como string (por exemplo, `next input = raw_input()`)

* Também é possível suprimir completamente a janela instantânea do console DOS para arquivos que recebem cliques no Windows. Os arquivos cujos nomes terminam com a extensão *.pyw* exibirão apenas janelas construídas por seu script e não a janela de console padrão do DOS. Os arquivos *.pyw* são simplesmente arquivos-fonte *.py* que têm esse comportamento operacional especial no Windows. Eles são usados principalmente por interfaces com o usuário desenvolvidas em Python que constróem suas próprias janelas e, freqüentemente, em conjunto com várias técnicas para salvar saída impressa e erros em arquivos.

- Suporta redirecionamentos de fluxo de entrada no nível do shell de sistema (por exemplo, `python spam.py < input.txt`), exatamente como a instrução `print` faz para saída.

Usaremos o `raw_input` de maneiras mais avançadas posteriormente neste texto. Consulte o Capítulo 10 para ver um exemplo de uso em um loop interativo.

Outras limitações do clique em ícones

Mesmo com o truque `raw_input`, dar um clique em ícones de arquivo tem seus perigos. Você também pode não conseguir ver *mensagens de erro* do Python. Se o seu script gera um erro, o texto da mensagem de erro é escrito na janela de console instantânea – a qual, então, desaparece imediatamente, como antes. Pior ainda, adicionar uma chamada de `raw_input` em seu arquivo não ajudará desta vez, pois seu script provavelmente será cancelado muito antes de chegar a essa chamada. Em outras palavras, você não poderá saber o que deu errado.

Por causa dessas limitações, provavelmente é melhor ver os cliques em ícones como uma maneira de executar programas após eles terem sido depurados. Principalmente ao começar, use outras técnicas, como as linhas de comando de sistema e o IDLE (posteriormente neste capítulo), para que você possa ver as mensagens de erro geradas e ver sua saída normal sem contar com truques de código. Quando estudarmos as exceções, posteriormente neste livro, aprenderemos também que é possível interceptar e se recuperar de erros, para que eles não terminem nossos programas. Fique alerta para a discussão sobre a instrução `try`, posteriormente neste livro, para ver uma maneira alternativa de impedir que a janela do console se feche em caso de erro.

IMPORTAÇÕES E RECARREGAMENTOS DE MÓDULO

Até aqui, estivemos falando de "módulos" de arquivo e usando a palavra "importar", sem explicar o que esses termos significam. Vamos estudar mais detalhamente os módulos e a arquitetura de programas maiores na Parte V. Como as importações também são uma maneira de executar programas, esta seção apresenta fundamentos suficientes sobre módulos, para você começar a aprender.

Em termos simples, todo arquivo de código Python cujo nome termina com a extensão *.py* é um *módulo*. Outros arquivos podem acessar os itens definidos por um módulo *importando* esse módulo; basicamente, as operações de importação carregam outro arquivo e garantem o acesso ao conteúdo do arquivo. Além disso, o conteúdo de um módulo torna-se disponível para o mundo exterior por meio de seus *atributos*, um termo que definiremos a seguir.

Esse modelo de serviços baseados em módulo é a idéia básica por trás da arquitetura de programas no Python. Normalmente, os programas maiores assumem a forma de vários arquivos de módulo, os quais importam ferramentas de outros arquivos de módulo. Um dos módulos é designado como principal, ou arquivo de nível superior, e é o que é executado para iniciar o programa inteiro.

Vamos nos aprofundar em tais questões sobre arquitetura posteriormente neste livro. Este capítulo está interessado, principalmente, no fato de que as operações de importação *executam*, como uma etapa final, o código de um arquivo que está sendo carregado. Por isso, importar um arquivo é uma outra maneira de executá-lo.

Por exemplo, se iniciarmos uma sessão interativa (no IDLE, a partir de uma linha de comando ou de outra forma), poderemos executar o arquivo *script4.py* original que apareceu anteriormente, com uma importação simples:

```
D:\LP-2ndEd\Examples>c:\python22\python
```

```
>>> import script4
win32
1267650600228229401496703205376
```

Por padrão, isso funciona, mas apenas uma vez por sessão (na realidade, processo). Após a primeira importação, as importações posteriores não fazem nada, mesmo que alteremos e salvemos o arquivo-fonte do módulo novamente, em outra janela:

```
>>> import script4
>>> import script4
```

Isso ocorre assim por design; as importações são operações dispendiosas demais para serem repetidas mais de uma vez por execução de programa. Conforme aprenderemos no Capítulo 15, as importações devem encontrar arquivos, compilar em código de byte e executá-lo. Se quisermos realmente obrigar o Python a executar o arquivo novamente na mesma sessão (sem parar e reiniciar a sessão), precisaremos, em vez disso, chamar a função interna `reload`:

```
>>> reload(script4)
win32
65536
<module 'script4' from 'script4.py'>
>>>
```

A função `reload` carrega e executa a versão corrente do código do seu arquivo, caso você o tenha alterado em outra janela. Isso permite que você edite e ponha em ordem o novo código dinamicamente, dentro da sessão interativa corrente do Python. Nessa sessão, por exemplo, a segunda instrução `print` em `script4` foi chamada em outra janela para imprimir `2 ** 16`, entre o tempo da primeira instrução `import` e a `chamada de reload`.

A função `reload` espera o nome de um objeto de módulo já carregado; portanto, você tem de ter importado com êxito uma vez, antes de recarregar. Note que `reload` também espera parênteses em torno do nome do objeto módulo, enquanto `import`, não espera – `reload` é uma função que é *chamada* e `import` é uma instrução. É por isso que devemos passar o nome do módulo como argumento entre parênteses, e é por isso que obtemos uma linha de saída extra ao recarregar. A última linha de saída é apenas a representação da impressão do valor de retorno da chamada de `reload`, um objeto módulo do Python. Mais informações sobre funções aparecem no Capítulo 12.

A grande história do módulo: atributos

As importações e os recarregamentos oferecem uma opção de execução de programa natural, pois os arquivos são executados pelas operações de importação como uma última etapa. Contudo, no esquema mais amplo das coisas, os módulos servem como *bibliotecas* de ferramentas, conforme aprenderemos na Parte V. Em geral, um módulo é principalmente apenas um pacote de nomes, conhecido como *espaço de nomes*. Além disso, os nomes dentro desse pacote são chamados de *atributos* – nomes de variável ligados a um objeto específico.

No uso típico, os importadores obtêm acesso a todos os nomes atribuídos no nível superior do arquivo de um módulo. Esses nomes normalmente são atribuídos aos serviços exportados pelo módulo – funções, classes, variáveis etc. –, os quais se destinam a ser usados em outros arquivos e em outros programas. Fora de um arquivo, os nomes podem ser buscados com duas instruções do Python, `import` e `from`, assim como com a chamada de `reload`.

Para ilustrar, suponha que usemos um editor de textos para criar o arquivo de módulo Python *myfile.py*, de apenas uma linha, mostrado no exemplo a seguir. Esse pode ser um dos módulos Python mais simples do mundo (ele contém uma única instrução de atribuição), mas é suficiente para ilustrar os fundamentos. Quando esse arquivo é importado, seu código é executado para gerar o atributo do módulo – a instrução de atribuição desse arquivo cria um atributo de módulo chamado `title`:

```
title = "The Meaning of Life"
```

Agora, podemos acessar o atributo `title` desse módulo em outros componentes, de duas maneiras diferentes. Com a instrução `import`, obtemos o módulo como um todo e *qualificamos* o nome do módulo pelo nome do atributo a acessar:

```
% python                          Inicia o Python.
>>> import myfile                 Executa o arquivo; carrega o módulo como um todo.
>>> print myfile.title            Usa seus nomes de atributo: '.' para qualificar.
The Meaning of Life
```

Em geral, a sintaxe de expressão do ponto `objeto.atributo` nos permite buscar qualquer atributo ligado a qualquer objeto e é uma operação comum em código Python. Aqui, a utilizamos para acessar a variável de string `title` dentro do módulo `myfile` – isto é, `myfile.title`. Como alternativa, podemos buscar (na realidade, copiar) nomes de um módulo com instruções `from`:

```
% python                          Inicia o Python.
>>> from myfile import title      Executa o arquivo; copia seus nomes.
>>> print title                   Usa o nome diretamente; não há necessidade de qualificar.
The Meaning of Life
```

Conforme veremos com mais detalhes posteriormente, `from` é exatamente como uma instrução `import`, com uma atribuição extra para nomes no componente de importação. Contudo, como ela também copia nomes do arquivo importado, usamos as importações diretamente, sem passar pelo nome do módulo original. Tecnicamente, `from` copia os *atributos* de um módulo, de modo que eles se tornam simples *variáveis* no destinatário – desta vez, nos referimos simplesmente à string importada como `title` (uma variável), em vez de `myfile.title` (uma referência de atributo).*

Seja usando `import` ou `from` para executar uma operação de importação, as instruções presentes no arquivo de módulo *myfile.py* são executadas e o componente de importação (aqui, o prompt interativo) obtém acesso aos nomes atribuídos no nível superior do arquivo. Existe apenas um deles neste exemplo simples – a variável `title`, atribuída a uma string –, mas o conceito será mais útil quando começarmos a definir objetos mais interessantes, como funções e classes, em nossos módulos. Tais objetos se tornam componentes de software reutilizáveis, acessados pelo nome a partir de um ou mais módulos cliente.

Na prática, os arquivos de módulo normalmente definem mais de um nome para ser usado dentro e fora do arquivo. Aqui está um exemplo que define três nomes:

```
a = 'dead'                        # Define três atributos.
b = 'parrot'                      # Exportado para outros arquivos
```

* Note que tanto `import` como `from` listam o nome do arquivo de módulo simplemente como *myfile*, sem o sufixo `.py`. Conforme aprenderemos na Parte 5, quando o Python procura o arquivo real, ele sabe incluir o sufixo no seu procedimento de pesquisa. Novamente, lembre-se de incluir o sufixo nas linhas de comando de shell do sistema, mas não em instruções de importação.

```
c = 'sketch'
print a, b, c                        # Também usado neste arquivo
```

Esse arquivo, *threenames.py*, atribui três variáveis e, assim, gera três atributos para o mundo exterior. Ele também usa suas próprias três variáveis em uma instrução `print`, conforme vemos ao executarmos isso como um arquivo de nível superior:

```
% python threenames.py
dead parrot sketch
```

Quando esse arquivo é importado em qualquer outra parte, todo o seu código é executado normalmente, como na primeira vez que ele é importado (por uma instrução `import` ou `from`). Os clientes desse arquivo que usam `import` recebem um módulo com atributos; os clientes que usam `from` recebem cópias dos nomes do arquivo:

```
% python
>>> import threenames                     Pega o módulo inteiro
dead parrot sketch
>>>
>>> threenames.b, threenames.c
('parrot', 'sketch')
>>>
>>> from threenames import a, b, c        Copia vários nomes
>>> b, c
('parrot', 'sketch')
```

Os resultados aqui são impressos entre parênteses, pois na verdade eles são *tuplas* – um tipo de objeto abordado na próxima parte deste livro.

Quando você começa a desenvolver módulos com vários nomes, como esse, a função interna `dir` começa a se tornar útil para buscar uma lista dos nomes disponíveis dentro de um módulo:

```
>>> dir(threenames)
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'b', 'c']
```

Quando a função `dir` é chamada com o nome de um módulo importado passado entre parênteses, como esse, ela retorna todos os atributos que estão dentro desse módulo. Alguns dos nomes retornados você recebe "gratuitamente": os nomes com sublinhados duplos no início e no fim, são nomes internos sempre predefinidos pelo Python e têm significado especial para o interpretador. As variáveis definidas pela atribuição em nosso código – a, b e c – aparecem por último no resultado de `dir`.

Notas sobre a utilização de import e reload

Por algum motivo, quando as pessoas descobrem que podem executar código por meio de importações e recarregamentos, muitas tendem a focalizar apenas isso e se esquecem das outras opções de execução que sempre executam a versão corrente do código (por exemplo, cliques em ícones, opções de menu do IDLE e linhas de comando). Isso pode causar confusão rapidamente – você precisa lembrar de quando tiver importado para saber se pode recarregar, precisa lembrar de usar parênteses no recarregamento (apenas) e precisa lembrar de usar reload, antes de tudo, para fazer a versão corrente do seu código ser executada.

Por causa dessas complicações (e outras que conheceremos posteriormente), evite a tentação de executar por meio de importações e recarregamentos, por enquanto. A opção de menu Edit/RunScript do IDLE, por exemplo, oferece uma maneira mais simples e menos propensa a erros para executar seus arquivos. Por outro lado, as importações e os recarregamentos têm

se mostrado uma técnica de teste popular em aulas sobre Python; portanto, você será o juiz. Contudo, se você achar que vai se dar mal, pare.

Há mais sobre a história dos módulos do que o exposto aqui, e você poderá ter problemas se utilizar módulos de maneiras incomuns neste ponto do livro; por exemplo, se você tentar importar um arquivo de módulo armazenado em um diretório que não seja aquele em que está trabalhando, terá que pular para o Capítulo 15 e aprender sobre o *caminho de pesquisa de módulo*. Por enquanto, se você precisar importar, tente manter todos os seus arquivos no diretório em que está trabalhando, para evitar complicações.

A INTERFACE COM O USUÁRIO IDLE

O IDLE é uma interface gráfica com o usuário para fazer desenvolvimento em Python, e é uma parte padrão e gratuita do sistema Python. Normalmente, ele é referido como IDE (*Integrated Development Environment* – ambiente de desenvolvimento integrado), pois reúne várias tarefas de desenvolvimento em um único modo de visualização.*

Em resumo, o IDLE é uma GUI que permite editar, executar, navegar e depurar programas em Python, tudo a partir de uma única interface. Além disso, como o IDLE é um programa em Python que utiliza o kit de ferramentas GUI Tkinter, ele é executado de forma portável na maioria das plataformas Python: MS Windows, X Windows (Unix, Linux) e Macs. Para muitas pessoas, o IDLE é uma alternativa fácil de usar para digitar linhas de comando e uma alternativa menos propensa a erros para clicar em ícones.

Fundamentos do IDLE

Vamos passar diretamente a um exemplo. É fácil iniciar o IDLE no Windows – ele tem uma entrada para o Python no menu do botão Iniciar (veja a Figura 2-1) e também pode ser selecionado dando-se um clique com o botão direito do mouse em um ícone de programa Python. Em alguns sistemas do tipo Unix, talvez você precise ativar o script de nível superior do IDLE a partir de uma linha de comando ou dar um clique no ícone – iniciar o arquivo `idle.pyw` no subdiretório `idle` do diretório `Tools` do Python.**

A Figura 3-3 mostra a cena após iniciar o IDLE no Windows. A janela Python Shell, na parte inferior, é a principal, a qual executa uma sessão interativa (observe o `prompt >>>`). Isso funciona como todas as sessões interativas – o código que você digita aqui é executado imediatamente após a digitação – e serve como ferramenta de teste.

O IDLE usa menus familiares, com atalhos de teclado para a maior parte de suas operações. Para fazer (ou editar) um script no IDLE, abra janelas de edição de textos – na janela principal, selecione o menu suspenso File e escolha New window para abrir uma janela de edição de textos (ou Open... para editar um arquivo já existente). A janela da parte superior da Figura 3-3 é uma janela de edição de textos do IDLE, onde o código do arquivo *script3.py* foi digitado.

Embora possa não aparecer detalhadamente neste livro, o IDLE usa *cores* designadas pela sintaxe para o código digitado na janela principal e em todas as janelas de edição de textos – as palavras-chave têm uma cor, as literais tem outra etc. Isso ajuda a proporcionar a você um quadro melhor dos componentes existentes em seu código.

* IDLE é uma corruptela de IDE e foi assim chamado em homenagem ao membro do grupo Monty Python, Eric Idle.

** O IDLE é um programa em Python que usa o kit de ferramentas GUI Tkinter da biblioteca padrão para construir a GUI IDLE. Isso torna o IDLE portável, mas também significa que você precisará ter suporte para Tkinter em seu Python para usar o IDLE. Por padrão, a versão para Windows do Python tem isso, mas os usuários de Linux e Unix ocasionalmente precisam instalar o suporte para Tkinter apropriado (veja as dicas de instalação no Apêndice A para conhecer os detalhes).

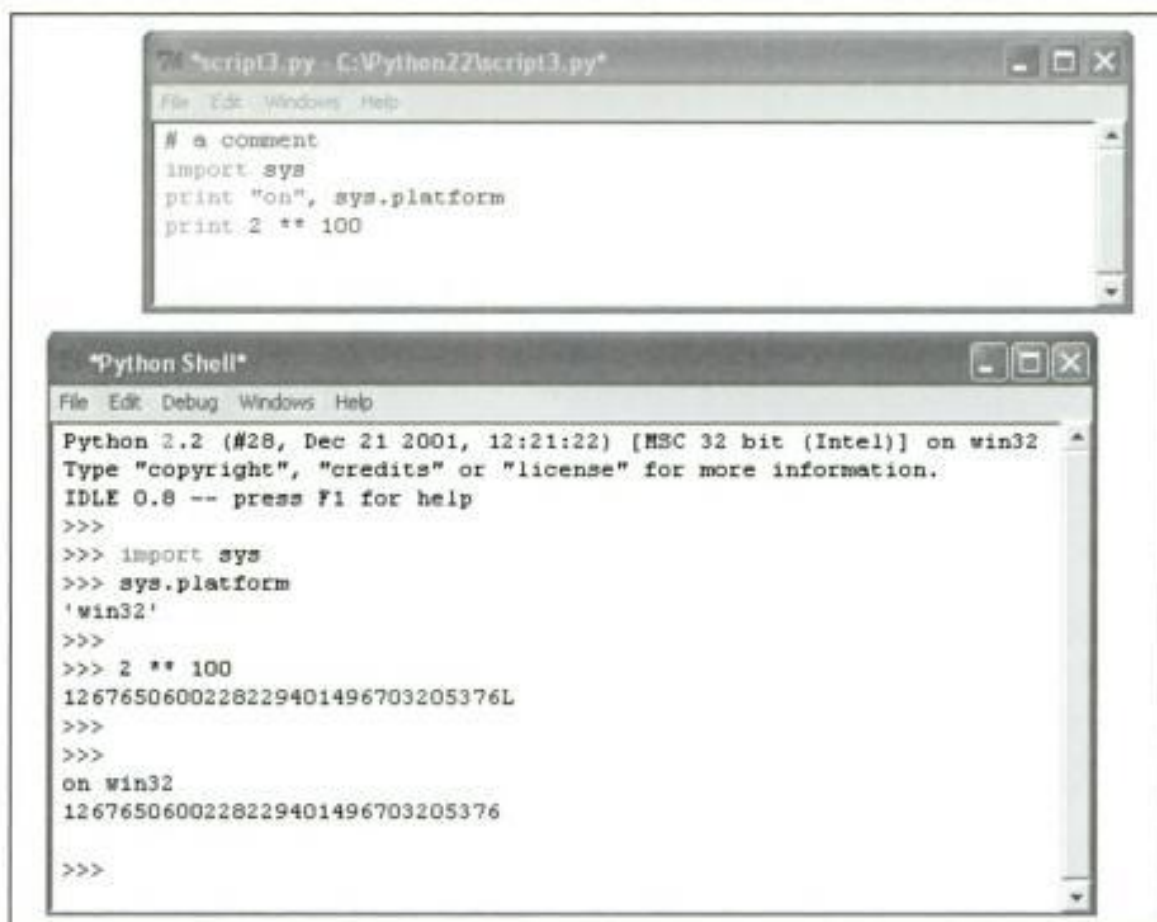



Figura 3-3 Janela principal e janela de edição de textos do IDLE.

Para executar um arquivo de código que você está editando no IDLE, selecione a janela de edição de textos do arquivo, escolha o menu suspenso `Edit` dessa janela e, nela, escolha a opção `Run Script` (ou use o atalho de teclado equivalente, listado no menu). O Python faz com que você saiba que precisa salvar seu arquivo primeiro, caso o tenha alterado desde que foi aberto ou salvo pela última vez. (No Python 2.3, a estrutura do menu muda ligeiramente; a opção Run Module no novo menu Run tem o mesmo efeito da seleção de menu Edit/Runscript da versão anterior. Veja o quadro "O IDLE muda na versão 2.3", posteriormente neste capítulo.)

Ao se executar dessa maneira, a saída do seu script e todas as mensagens de erro que ele possa gerar aparecem na janela interativa principal. Na Figura 3-3, por exemplo, as duas últimas linhas na janela interativa inferior refletem uma execução do script que está na janela de edição de textos superior.

Dica do dia: para repetir comandos anteriores na janela interativa principal do IDLE, use Alt-P para rolar para trás no histórico de comandos e Alt-N para rolar para frente. Seus comandos anteriores são lembrados e exibidos, e podem ser editados e executados novamente. Você também pode re-executar comandos posicionando o cursor sobre eles ou usar operações de recortar e colar, mas isso tende a dar mais trabalho. Fora do IDLE, você pode re-executar comandos em uma sessão interativa com as teclas de seta no Windows, se estiver executando doskey ou usando uma versão recente do Windows.

Usando o IDLE

O IDLE é gratuito, fácil de usar e portável. Mas também é um tanto limitado, comparado aos IDEs comerciais mais avançados. Aqui está uma lista dos problemas comuns que parecem atrapalhar os iniciantes no IDLE:

Você deve adicionar ".py" explicitamente ao salvar seus arquivos. Mencionamos isso quando falamos sobre arquivos em geral, mas esse é um obstáculo comum do IDLE, especialmente para usuários de Windows. O IDLE não adiciona a extensão *.py* automaticamente nos nomes de arquivo, quando os arquivos são salvos. Tome o cuidado de digitar você mesmo a extensão *.py* no final de nomes de arquivos, quando salvá-los pela primeira vez. Se você não fizer isso, poderá executar seus arquivos a partir do IDLE (e de linhas de comando), mas não poderá importá-los interativamente nem a partir de outros módulos.

Certifique-se de não estar vendo mensagens de erro antigas. Atualmente, o IDLE não faz um bom trabalho de separar a saída de cada execução de script na janela interativa – não há nenhuma linha em branco entre as saídas. Por isso, muitos iniciantes podem ser enganados por uma mensagem de erro de uma execução anterior, pensando que seus scripts ainda estão falhando, quando, na verdade, estão funcionando bem, porém em silêncio. Certifique-se de que as mensagens de erro que você ver não sejam antigas – digitar uma linha vazia na janela interativa ajuda.

Execute scripts a partir de janelas de edição de textos e não da janela interativa. Para executar um arquivo de código no IDLE, sempre selecione a opção de menu Edit/RunScript dentro da *janela de edição de textos*, onde você está editando o código a ser executado – e não dentro da janela interativa principal, onde aparece o prompt `>>>`. A opção RunScript nem mesmo deve estar disponível na janela interativa (e, na verdade, parece ter desparecido na versão recente); se você selecioná-la, acabará tentando executar um log da sua sessão interativa, com resultados bastante indesejáveis!

Os programas da GUI Tkinter podem não funcionar bem com o IDLE. Como o IDLE é um programa Python/Tkinter, pode travar se você usá-lo para executar certos tipos de programas Python/Tkinter, especialmente se seu código executar uma chamada `mainloop` da Tkinter. Seu código pode não apresentar tais problemas, mas como regra básica, será sempre seguro se você usar o IDLE para editar programas de GUI, mas executá-los usando outras opções, como cliques ou linhas de comando.

Outros programas também podem não funcionar bem com o IDLE. Em geral, como o IDLE atualmente (na versão 2.2) executa seu código no mesmo processo em que o próprio IDLE é executado, não é impossível travá-lo também com programas que não são GUIs. Na verdade, como o IDLE utiliza processos, e não segmentos, para executar scritps, um loop infinito pode torná-lo completamente impassível. Como regra básica, se você não puder encontrar um motivo para uma falha de programa sob o IDLE, tente executá-lo fora do IDLE para certificar-se de que seu problema não é, na verdade, um problema de IDLE.

Isso pode melhorar com o passar do tempo (veja o quadro "O IDLE muda na versão 2.3"). O inconveniente dessa estrutura atualmente é que, como seu script é executado no ambiente do IDLE, as variáveis em seu código aparecem automaticamente na sessão interativa – você nem sempre precisa executar comandos de importação para acessar nomes no nível superior de arquivos que já executou.

Execute scripts por meio de Edit/Run Script e não de importações e recarregamentos. Na seção anterior, vimos que também é possível executar um arquivo importando-o interati-

vamente. Entretanto, esse esquema pode ficar complexo, pois você é obrigado a recarregar arquivos manualmente, após fazer alterações. Em contraste, a opção Edit/RunScript no IDLE sempre executa a versão mais atual do seu arquivo. Ela também o avisa para salvá-lo primeiro, se necessário – outro erro comum fora do IDLE.

Tecnicamente falando, a opção Edit/Runscript do IDLE sempre executa a versão corrente apenas do arquivo de nível superior; talvez ainda seja preciso recarregar interativamente os arquivos importados, quando forem alterados. Em geral, contudo, a opção Edit/Runscript elimina confusões comuns que rodeiam as importações. Se, em vez disso, você optar por usar a técnica de importação e recarregamento, lembre-se de usar as combinações de tecla Alt-P/Alt-N para re-executar comandos anteriores.

Personalizando o IDLE. Para alterar as fontes de texto e cores no IDLE, edite os arquivos de configuração no diretório-fonte do IDLE. Por exemplo, para o Python 2.2 no Windows, o arquivo `C:\Python22\Tools\idle\config-win.txt` especifica fonte de texto e tamanho. Consulte o arquivo *config.txt* nesse diretório ou o menu suspenso de ajuda do IDLE para ver mais dicas. Consulte também o quadro "O IDLE muda na versão 2.3".

Atualmente, não existe nenhuma opção para limpar a tela no IDLE. Isso parece ser um pedido freqüente (talvez por causa de IDEs semelhantes) e, eventualmente, pode ser adicionado. Atualmente, contudo, não há nenhuma maneira de limpar o texto da janela interativa. Se você quiser que o texto da janela desapareça, pode manter a tecla Enter pressionada ou digitar um loop em Python para imprimir linhas em branco.

Além das funções de edição e execução básicas, o IDLE fornece recursos mais avançados, incluindo um depurador de programas do tipo apontar e clicar e um navegador de objetos. A Figura 3-4, por exemplo, mostra as janelas do depurador e do navegador de objetos do IDLE em ação. O navegador permite que você percorra o caminho de pesquisa de módulo para arquivos e objetos em arquivos; clicar em um arquivo ou objeto abre o código-fonte correspondente em uma janela de edição de textos.

O IDLE muda na versão 2.3

A maioria dos alertas desta seção sobre o IDLE torna-se discutível na nova versão, o Python 2.3. Uma nova versão do IDLE, conhecida inicialmente como IDLEfork, foi incorporada no Python 2.3 após esta seção ter sido escrita. Essa nova versão corrige muitas das limitações da versão anterior.

Por exemplo, o novo IDLE executa seu código em um processo separado, de tal modo que não atrapalha o IDLE em si. Esse processo separado é reiniciado para cada execução iniciada a partir de uma janela de edição de texto. Por isso, os problemas de interferência ao executar GUIs ou outro código dentro do IDLE devem ser uma coisa do passado.

Além disso, o novo IDLE muda a estrutura de menu, de modo que a opção de menu Edit/Run Script original está agora em Run/Run Module e só aparece na janela de edição de texto (não é mais possível executar inadvertidamente o texto da sessão interativa). O novo IDLE também inclui ferramentas de configuração baseadas em GUI, as quais permitem que fontes e cores sejam personalizadas sem a edição de arquivos de texto, e faz um trabalho muito melhor de separar a mensagem de erro de uma execução de um programa do próximo. Para obter mais detalhes, experimente por conta própria a versão 2.3 do IDLE.

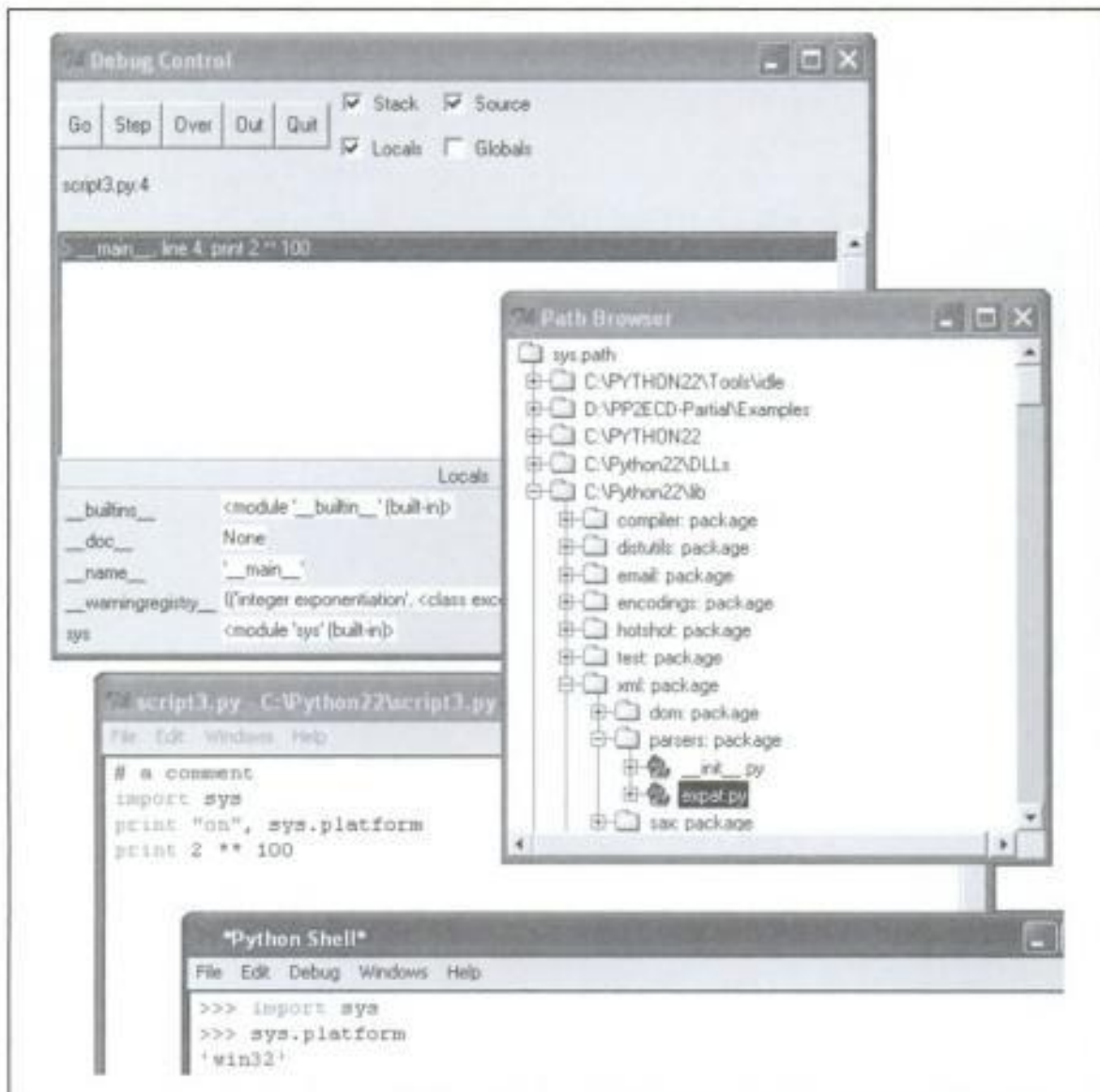



Figura 3-4 O depurador e o navegador de caminho/objeto do IDLE.

A depuração do IDLE é iniciada selecionando-se a opção de menu Debug/Debugger na janela principal e, então, iniciando seu script, selecionando-se a opção Edit/Run Script na janela de edição de texto; uma vez iniciada, você pode configurar pontos de interrupção em seu código, que interrompem a execução por meio de um clique com o botão direito do mouse nas linhas das janelas de edição de texto, mostram valores de variáveis etc. Você também pode observar a execução do programa ao depurar – selecionar a alternância de código-fonte do depurador fará a linha ativa ser destacada na janela de edição de texto, à medida que você percorre seu código.

Além disso, o editor de textos do IDLE oferece um grande conjunto de ferramentas amigáveis para o programador, incluindo endentação automática, operações de pesquisa de texto e arquivo avançadas e muito mais. Como o IDLE utiliza interações de GUI intuitivas, experimente o sistema ao vivo para ter uma idéia de suas outras ferramentas.

OUTROS IDES

Como o IDLE é gratuito, portável e uma parte padrão do Python, é uma primeira ferramenta de desenvolvimento interessante para se familiarizar, se você quiser usar um IDE. Use o

IDLE para os exercícios deste livro, se estiver apenas começando. Entretanto, existem muitos outros IDEs alternativos para desenvolvedores de Python, alguns dos quais são significativamente mais poderosos e robustos do que o IDLE. Dentre os mais comumente usados atualmente estão estes quatro:

Komodo

Uma GUI de ambiente de desenvolvimento completa para o Python (e outras linguagens). A Komodo inclui edição de texto com cor padrão para sintaxe, depuração etc. Além disso, ela oferece muitos recursos avançados que o IDLE não oferece, incluindo arquivos de projeto, integração de controle de código-fonte, depuração de expressões regulares e um construtor de GUI, tipo arrastar e soltar, que gera código Python/Tkinter para implementar as GUIs que você projeta interativamente. A Komodo não era gratuita, quando escrevemos este livro; ela está disponível no endereço *http://www.activestate.com*.

PythonWorks

Outra GUI de ambiente de desenvolvimento completa. A PythonWorks também tem ferramentas de IDE padrão e fornece um construtor de GUI Python/Tkinter que gera código Python. Além disso, suporta recursos únicos, como refabricação de código automática para manutenção e reutilização otimizadas. Este também é um produto comercial; consulte o endereço *http://www.pythonware.com* para ver os detalhes.

PythonWin

Um IDE gratuito que vem como parte da distribuição ActivePython da ActiveState (e também pode ser procurado separadamente a partir de recursos do endereço *http://www.python.org*). O PythonWin é um IDE do Python somente para Windows; ele é praticamente como o IDLE, com muitas extensões úteis específicas para Windows incorporadas. Por exemplo, o PythonWin tem suporte para objetos COM. Ele também acrescenta mais recursos básicos de interface com o usuário do que o IDLE, como menus instantâneos de lista de atributos de objeto. Além disso, o PythonWin serve como exemplo de uso da biblioteca de GUI do pacote de extensão Windows. Consulte o endereço *http://www.activestate.com*.

Visual Python

A ActiveState também comercializa um sistema chamado Visual Python, que é um plugin que adiciona suporte para Python no ambiente de desenvolvimento Visual Studio da Microsoft. Essa também é uma solução somente para Windows, mas é atraente para desenvolvedores com conhecimento intelectual anterior em Visual Studio. Consulte o site da Web da ActiveState para ver os detalhes.

Conhecemos aproximadamente mais uma meia dúzia de outros IDEs (por exemplo, WingIDE, PythonCard) e provavelmente mais aparecerão com o passar do tempo. Em vez de tentar documentar todos eles aqui, consulte os recursos disponíveis no endereço *http://www.python.org*, assim como o site da Web Vaults of Parnassus.

INCORPORANDO CHAMADAS

Até aqui, vimos como se faz para executar o código digitado interativamente e como executar código salvo em arquivos com linhas de comando, cliques em ícones, IDEs, importações de módulo e scripts executáveis do Unix. Isso cobre a maioria dos casos que veremos neste livro.

Mas em alguns domínios especializados, o código Python também pode ser executado por um sistema fechado. Nesses casos, dizemos que os programas em Python são *incorporados*

em outro programa (isto é, executados por este). O código Python, em sessão interativa, pode ser inserido em um arquivo de texto, armazenado em um banco de dados, buscado de uma página em HTML, analisado a partir de um documento em XML etc. Mas, da perspectiva operacional, outro sistema – e não você – pode dizer ao Python para que execute o código que você criou.

Por exemplo, é possível criar e executar strings de código Python a partir de um programa em C, chamando funções na API de tempo de execução do Python (um conjunto de serviços exportados pelas bibliotecas criadas quando o Python é compilado em sua máquina):

```
#include <Python.h>
...
Py_Initialize();
PyRun_SimpleString("x = brave + sir + robin");
```

Nesse trecho de código em C, um programa desenvolvido na linguagem C incorpora o interpretador do Python vinculando em suas bibliotecas, e o passa para execução em uma string de instrução de atribuição Python. Os programas em C também podem ter acesso aos objetos Python e processá-los ou executá-los usando outras ferramentas de API Python.

Este livro não é sobre a integração Python/C, mas você deve saber que, dependendo de como sua empresa pretende usar o Python, pode ou não ser o responsável por iniciar os programas em Python que criar.* Independente disso, você provavelmente ainda pode usar as técnicas de execução interativa e baseadas em arquivo, descritas aqui, para testar o código isoladamente dos sistemas fechados que possam eventualmente utilizá-lo.

BINÁRIOS CONGELADOS EXECUTÁVEIS

Os binários congelados executáveis são pacotes que combinam o código de byte do seu programa e o interpretador do Python em um único programa executável. Com eles, os programas podem ser executados da mesma maneira como você faria para executar outro programa executável (cliques em ícones, linhas de comando etc.). Embora essa opção funcione bem para a distribuição de produtos, na verdade não se destina a ser usada durante o desenvolvimento de programas. Normalmente, você congela apenas antes de distribuir e depois que o desenvolvimento está concluído.

OPÇÕES DE EXECUÇÃO COM EDITOR DE TEXTOS

Muitos editores de textos, amigáveis para o programador, têm suporte para edição (e possivelmente execução) de programas em Python. Tal suporte pode ser incorporado ou buscado na Web. Por exemplo, se você estiver familiarizado com o editor de textos emacs, pode fazer toda sua edição e execução do Python dentro do próprio editor. Consulte a página de recursos de editor de textos no endereço *http://www.python.org/editors* para obter mais detalhes.

OUTRAS OPÇÕES DE EXECUÇÃO

Dependendo de sua plataforma, podem existir mais maneiras de iniciar programas em Python. Por exemplo, em alguns sistemas Macintosh, você pode arrastar ícones de arquivo de programa em Python para o ícone do interpretador da linguagem, para fazê-los serem executados.

* Consulte o livro *Programming Python* (O'Reilly) para ver mais detalhes sobre a incorporação do Python em C/C++. A API de incorporação pode chamar funções Python diretamente, carregar módulos e muito mais. Note também que o sistema Jython permite que programas em Java executem código Python, usando uma API baseada em Java (uma classe de interpretador Python).

E, no Windows, você sempre pode iniciar scripts Python com a opção Executar... no menu Iniciar. Finalmente, a biblioteca padrão do Python tem utilitários que permitem aos programas em Python serem iniciados por outros programas escritos nessa linguagem (por exemplo, `execfile, os.popen, os.system`); entretanto, essas ferramentas estão fora dos objetivos deste capítulo.

POSSIBILIDADES FUTURAS?

Embora este capítulo reflita a prática corrente, grande parte dele foi específica para uma plataforma ou uma época específica. Na verdade, muitos detalhes da execução e da inicialização apresentados surgiram entre a primeira e a segunda edições deste livro. Com relação à execução de programas, não é impossível que novas opções de execução possam surgir com o passar do tempo.

Novos sistemas operacionais e novas versões deles também podem fornecer técnicas de execução além daquelas delineadas aqui. Em geral, como o Python acompanha tais mudanças, você vai conseguir executá-lo da maneira que fizer sentido para as máquinas que utilizar, tanto agora como no futuro – seja desenhando em PCs de mesa gráfica ou PDAs, pegando ícones em um equipamento de realidade virtual ou pronunciando o nome de um script em conversas com seus colegas.

Mudanças na implementação também podem ter impacto nos esquemas de ativação (por exemplo, um compilador completo poderia produzir executáveis normais, ativados como os binários congelados de hoje). Contudo, se soubéssemos o que o futuro realmente reserva, provavelmente estaríamos falando para um corretor da Bolsa e não escrevendo isto.

QUAL OPÇÃO DEVO USAR?

Com todas essas opções, a pergunta surge naturalmente: qual é a melhor para mim? Em geral, use a interface do IDLE para desenvolvimento, se estiver começando a usar o Python. Ela fornece um ambiente de GUI amigável e pode ocultar parte dos detalhes subjacentes da configuração. Ela também vem com um editor de textos neutro quanto à plataforma para desenvolver seus scripts, e é uma parte padrão e gratuita do sistema Python.

Se, em vez disso, você é um programador experiente, talvez se sinta mais à vontade simplesmente com o editor de textos de sua escolha em uma janela, e outra janela para executar os programas que edita, por meio de linhas de comando ou cliques em ícones. Como os ambientes de desenvolvimento são uma escolha muito subjetiva, não podemos oferecer muito mais opções como diretrizes universais; em geral, o ambiente que você opta por usar normalmente é o melhor.

EXERCÍCIOS DA PARTE I

É hora de você mesmo começar a desenvolver um pouco. Esta primeira sessão de exercícios é muito simples, mas algumas destas questões sugerem assuntos que aparecerão em capítulos posteriores. Lembre-se: consulte o Apêndice A para ver as respostas. Às vezes, os exercícios e suas soluções contêm informações suplementares não discutidas na parte principal do capítulo. Em outras palavras, você deve dar uma olhada, mesmo que consiga dar todas as respostas sozinho.

1. *Interação*. Usando uma linha de comando de sistema, IDLE ou outro, inicie a linha de comando interativa do Python (prompt `>>>`) e digite a expressão: `"Hello World!"`

(incluindo as aspas). A string deve ser ecoada de volta para você. O objetivo deste exercício é configurar seu ambiente para executar o Python. Em alguns cenários, talvez você precise primeiro executar o comando de shell `cd`, digitar o caminho completo até o executável `python` ou adicionar seu caminho na variável de ambiente PATH. Se quiser, você pode configurá-la em seu arquivo *.cshrc* ou*.kshrc* para tornar o Python permanentemente disponível em sistemas Unix; no Windows, use *setup.bat*, *autoexec.bat* ou a GUI de variável de ambiente. Consulte o Apêndice A para obter ajuda nas configurações de variável de ambiente.

2. *Programas*. Com o editor de textos de sua escolha, escreva um arquivo de módulo simples – um arquivo contendo uma única instrução: `print 'Hello module world!'`. Armazene essa instrução em um arquivo chamado *module1.py*. Agora, execute esse arquivo usando a opção de execução que desejar: executando-o no IDLE, dando um clique no ícone de seu arquivo, passando-o para o programa interpretador do Python na linha de comando de shell do sistema etc. Na verdade, experimente executar seu arquivo com o máximo das técnicas de execução vistas neste capítulo que você puder. Qual técnica parece mais fácil? (Não existe uma única resposta certa para esta pergunta.)
3. *Módulos*. Em seguida, inicie a linha de comando interativa do Python (prompt `>>>`) e importe o módulo que você escreveu no Exercício 2. Tente mover o arquivo para um diretório diferente e importá-lo novamente a partir de seu diretório original (isto é, execute o Python no diretório original quando você importar); o que acontece? (Dica: ainda existe um arquivo chamado *module1.pyc* no diretório original?)
4. *Scripts*. Se sua plataforma suporta isso, adicione a linha #! no início do seu módulo `module1.py`, dê ao arquivo privilégios de executável e execute-o diretamente. O que a primeira linha precisa conter? Pule isso, caso você esteja trabalhando em uma máquina Windows (`#!` normalmente só tem significado no Unix e no Linux); em vez disso, tente executar seu arquivo listando apenas seu nome em uma janela de console DOS (isso funciona nas versões recentes do Windows) ou na caixa de diálogo Iniciar/Executar...
5. *Erros*. Experimente digitar expressões matemáticas e atribuições na linha de comando interativa do Python. Primeiro, digite a expressão `1/0`. O que acontece? Em seguida, digite um nome de variável para a qual você ainda não atribuiu um valor. O que acontece desta vez?

   Você pode ainda não saber, mas está realizando processamento de exceção, um assunto que exploraremos com profundidade na Parte VII. Conforme vai aprender lá, tecnicamente você está executando o que é conhecido como *rotina de tratamento de exceção padrão* – lógica que imprime uma mensagem de erro padrão.

   Para tarefas de *depuração* de código-fonte completas, o IDLE inclui uma interface de depuração introduzida neste capítulo (veja a seção sobre utilização avançada do IDLE), e um módulo da biblioteca padrão do Python chamado `pdb` fornece uma interface de depuração de linha de comando (mais informações sobre `pdb` aparecem no manual da biblioteca padrão). Quando iniciadas pela primeira vez, as mensagens de erro padrão do Python provavelmente serão a maior parte do tratamento de erros que você precisa – elas indicam a causa do erro e também mostram as linhas que estavam ativas em seu código quando o erro ocorreu.
6. *Interrupções*. Na linha de comando do Python, digite:

   ```
   L = [1, 2]
   L.append(L)
   L
   ```

O que acontece? Se você estiver usando uma versão do Python mais recente do que a 1.5, provavelmente verá uma saída estranha, que descreveremos na próxima parte do livro. Se você estiver usando uma versão do Python mais antiga do que a 1.5.1, uma combinação de teclas Ctrl-C provavelmente ajudará na maioria das plataformas. Por que você acha que isso ocorre? O que o Python relata quando você digita a combinação de teclas Ctrl-C? Alerta: se você tem uma versão do Python mais antiga do que a 1.5.1, antes de fazer este teste, certifique-se de que sua máquina possa interromper um programa com uma combinação break-tecla de algum tipo, senão você poderá esperar um longo tempo.

7. *Documentação*. Antes de prosseguir, passe pelo menos 17 minutos examinando a biblioteca do Python e os manuais da linguagem, para ter uma idéia das ferramentas disponíveis na biblioteca padrão e da estrutura do conjunto de documentos. Demora no mínimo esse tempo para familiarizar-se com a localização dos principais tópicos no conjunto de manuais. Quando você terminar, será fácil encontrar o que precisa. Você pode encontrar esse manual na entrada do botão Iniciar para o Python no Windows, no menu suspenso Help no IDLE ou on-line, no endereço *http://www.python.org/docs*. Também teremos mais algumas palavras a dizer sobre os manuais e outras fontes de documentação disponíveis (incluindo PyDoc e a função `help`), no Capítulo 11. Se ainda tiver tempo, explore o site na Web do Python (*http://www.python.org*) e o link do site Vault of Parnassus que encontrará lá. Consulte especialmente a documentação presente no site python.org e as páginas de pesquisa. Na prática, elas podem ser recursos decisivos.

Tipos e Operações

Na Parte II, estudaremos os tipos de dados básicos internos do Python, às vezes chamados de tipos de objeto. Embora existam mais tipos de objeto no Python do que encontraremos nesta parte, os tipos discutidos aqui geralmente são considerados tipos de dados básicos – os principais temas de praticamente todo script Python que você provavelmente lê ou escreve.

Esta parte do livro está organizada em torno dos principais tipos básicos, mas fique atento para assuntos relacionados, como tipagem dinâmica e categorias de objeto gerais, que também aparecem pelo caminho. Como os capítulos desta parte formam a base pressuposta para os capítulos posteriores, ela funcionará melhor se for lida de maneira linear.

4

Números

Este capítulo inicia nosso tour pela linguagem Python. No Python, os dados assumem a forma de objetos – objetos incorporados fornecidos pelo Python ou objetos que criamos usando ferramentas Python e C. Como os objetos são a noção mais fundamental na programação em Python, vamos começar este capítulo com um levantamento dos tipos de objeto internos da linguagem, antes de nos concentrarmos nos números.

ESTRUTURA DE PROGRAMA EM PYTHON

Como introdução, vamos primeiro mostrar claramente como este capítulo se encaixa no quadro geral do Python. A partir de uma perspectiva mais concreta, os programas em Python podem ser decompostos em módulos, instruções, expressões e objetos, conforme segue:

1. Os programas são compostos de módulos.
2. Os módulos contêm instruções.
3. As instruções contêm expressões.
4. As expressões criam e processam objetos.

Nós apresentamos o nível mais alto dessa hierarquia quando aprendemos a respeito de módulos, no Capítulo 3. Os capítulos desta parte começam na parte inferior, explorando os *objetos* incorporados e as *expressões* que você pode escrever para usá-los.

POR QUE USAR TIPOS INCORPORADOS?

Se você já usou linguagens de nível mais baixo, como C ou C++, então sabe que grande parte de seu trabalho concentra-se na implementação de *objetos* – também conhecidos como *estruturas de dados* – para representar os componentes no domínio de sua aplicação. Você precisa organizar as estruturas de memória, gerenciar a alocação da memória, implementar rotinas de pesquisa e acesso etc. Essas tarefas são tão maçantes (e propensas a erros) quanto parecem e, normalmente, se desviam dos objetivos reais de seus programas.

Nos programas em Python típicos, a maior parte desse trabalho chato desaparece. Como o Python fornece tipos de dados poderosos como uma parte intrínseca da linguagem, não há ne-

cessidade de você escrever implementações de objeto antes de começar a resolver problemas. Na verdade, a não ser que você tenha necessidade de processamento especial que os tipos internos não fornecem, é quase sempre melhor usar um objeto incorporado do que implementar o seu próprio. Aqui estão alguns motivos disso:

Os objetos incorporados facilitam a escrita de programas simples. Para tarefas simples, os tipos internos são, freqüentemente, tudo que você precisa para representar a estrutura dos domínios do problema. Como você recebe coisas como coleções (listas) e tabelas de pesquisa (dicionários) gratuitamente, pode usá-las imediatamente. Você pode fazer muitas coisas apenas com os tipos de objeto incorporados do Python.

O Python fornece objetos e suporta extensões. De certa maneira, o Python pede emprestado tanto de linguagens que contam com ferramentas internas (por exemplo, LISP) como de linguagens que contam com o programador, para fornecer suas próprias implementações de ferramenta ou frameworks (por exemplo, C++). Embora você possa implementar tipos de objeto exclusivos em Python, não precisa fazer isso apenas para começar a trabalhar. Além disso, como os tipos internos do Python são padronizados, eles são sempre os mesmos; os frameworks tendem a diferir de um lugar para outro.

Os objetos incorporados são componentes de extensões. Para tarefas mais complexas, talvez você ainda precise fornecer seus próprios objetos, usando instruções do Python ou interfaces da linguagem C. Mas, conforme veremos em partes posteriores, freqüentemente os objetos implementados manualmente são construídos sobre tipos internos, como listas e dicionários. Por exemplo, uma estrutura de dados de pilha pode ser implementada como uma classe que gerencia uma lista interna.

Freqüentemente, os objetos incorporados são mais eficientes do que as estruturas de dados personalizadas. Os tipos internos do Python empregam algoritmos de estrutura de dados já otimizados, que são implementados em C por causa da velocidade. Embora você possa escrever seus próprios tipos de objeto semelhantes, normalmente estará sem tempo para obter o nível de desempenho fornecido pelos tipos de objeto incorporados.

Em outras palavras, os tipos de objeto incorporados não apenas tornam a programação mais fácil, como também são mais poderosos e eficientes do que a maioria que possa ser criada a partir do zero. Independente de você implementar novos tipos de objeto ou não, os objetos incorporados formam a base de todo programa em Python.

A Tabela 4-1 fornece uma prévia dos tipos de objeto incorporados e parte da sintaxe usada para escrever suas *literais* – expressões que geram objetos.* Alguns desses tipos provavelmente parecerão familiares, caso você já tenha usado outras linguagens. Por exemplo, os números e as strings representam valores numéricos e textuais, respectivamente, e os arquivos fornecem uma interface para processar arquivos armazenados em seu computador.

Os tipos de objeto da Tabela 4-1 são mais gerais e poderosos do que aqueles com que você pode estar acostumado. Por exemplo, você verá que listas e dicionários evitam a maior parte do trabalho que você pode fazer para suportar coleções e pesquisa em linguagens de nível mais baixo. As listas são coleções ordenadas de outros objetos e indexadas por posições que começam em 0. Os dicionários são coleções de outros objetos que são indexados pela chave,

---

* Neste livro, o termo *literal* significa simplesmente uma expressão cuja sintaxe gera um objeto – às vezes também chamado de *constante*. Se você ouvir esse objeto chamado de constante, não quer dizer que os objetos ou as variáveis nunca possam ser alterados (isto é, não tem relação com o tipo `const` da linguagem C++ ou com "imutável" do Python – um assunto que vai ser explorado posteriormente nesta parte do livro).

Tabela 4-1 Prévia dos objetos incorporados

Tipo de objeto	Exemplo de literais/criação
Números	`3,1415, 1234, 999L, 3+4j`
Strings	`'spam', "guido's"`
Listas	`[1, [2, 'three'], 4]`
Dicionários	`{'food': 'spam', 'taste': 'yum'}`
Tuplas	`(1, 'spam', 4, 'U')`
Arquivos	`text = open('eggs', 'r').read()`

em vez da posição. Tanto os dicionários quanto as listas podem ser aninhados, podem crescer e diminuir de acordo com a demanda e podem conter objetos de qualquer tipo. Contudo, para saber a história completa, você terá que continuar lendo.

NÚMEROS

O primeiro tipo de objeto do nosso passeio são os números do Python. Os tipos numéricos do Python são bastante característicos e parecerão familiares, caso você já tenha usado praticamente qualquer outra linguagem de programação. Eles podem ser usados para controlar seu saldo bancário, a distância até Marte, o número de visitantes em seu site na Web e, praticamente, qualquer outra quantidade numérica.

O Python suporta os tipos numéricos normais (conhecidos como *inteiro* e *ponto flutuante*), assim como literais, para a criação de números e expressões para processá-los. Além disso, o Python fornece suporte para programação numérica mais avançada, incluindo um tipo de número complexo, um inteiro de precisão ilimitada e uma variedade de bibliotecas de ferramentas numéricas. As próximas seções fornecem uma visão geral do suporte numérico no Python.

Literais numéricas

Dentre seus tipos básicos, o Python suporta os tipos numéricos normais: números inteiros e ponto flutuante, e toda a sua sintaxe e operações associadas. Assim como a linguagem C, o Python também permite que você escreva inteiros usando literais em hexadecimal e octal. Ao contrário da linguagem C, o Python também tem um tipo de número complexo, assim como um tipo inteiro longo com precisão ilimitada (ele pode crescer até ter tantos dígitos quanto seu espaço em memória permitir). A Tabela 4-2 mostra como são os tipos numéricos do Python, quando escritos em um programa (isto é, como literais).

Tabela 4-2 Literais numéricas

Literal	Interpretação
`1234, -24, 0`	Inteiros normais (valores long da linguagem C)
`9999999999999999999L`	Inteiros longos (tamanho ilimitado)
`1.23, 3.14e-10, 4E210, 4.0e+210`	Ponto flutuante (valores double da linguagem C)
`0177, 0x9ff, 0XFF`	Literais em octal e em hexadecimal
`3+4j, 3.0+4.0j, 3J`	Literais numéricas complexas

Em geral, os tipos numéricos do Python são simples e diretos, mas é interessante destacar aqui alguns conceitos de desenvolvimento:

Literais inteiros e ponto flutuante

Os inteiros são escritos como uma string de dígitos decimais. Os números de ponto flutuante têm um ponto decimal incorporado e/ou um expoente com sinal opcional, introduzido por `e` ou `E`. Se você escrever um número com um ponto decimal ou um expoente, o Python o transformará em um objeto de tipo ponto flutuante e utilizará matemática de ponto flutuante (não de inteiros) quando ele for usado em uma expressão. As regras para escrever números de ponto flutuante são as mesmas da linguagem C.

Precisão numérica e inteiros longos

Os inteiros puros do Python (linha 1 da Tabela 4-2) são implementados internamente como valores "long" da linguagem C (isto é, pelo menos 32 bits), e os números de ponto flutuante do Python são implementados como valores "double" da linguagem C; os números do Python obtêm a mesma precisão que o compilador C – usado para construir o interpretador do Python – fornece para valores long e double.*

Literais de inteiro longo

Por outro lado, se um literal inteiro termina com `l` ou `L`, ele se torna um inteiro longo do Python (não confundir com um valor long da linguagem C) e pode crescer o quanto for necessário. No Python 2.2, como os inteiros são convertidos em inteiros longos em caso de estouro de pilha, a letra `L` não é mais rigorosamente exigida.

Literais em hexadecimal e octal

As regras para escrever inteiros em hexadecimal (base 16) e em octal (base 8) são as mesmas da linguagem C. Os literais em octal começam com um `zero (0)`, seguido de uma string de dígitos `0-7`; os hexadecimais começam com `0x` ou `0X`, seguido dos dígitos em hexadecimal `0-9` e `A-F`. Nos literais em hexadecimal, os dígitos podem ser escritos em letras minúsculas ou maiúsculas.

Números complexos

Os literais complexos do Python são escritos como `parte_real+parte_imaginária`, onde a `parte_imaginária` termina com `j` ou `J`. Tecnicamente, a `parte_real` é opcional e a `parte_imaginária` pode vir primeiro. Internamente, elas são implementadas como um par de números em um ponto flutuante, mas todas as operações numéricas efetuam operações de números complexos, quando aplicadas a números complexos.

Ferramentas internas e extensões

Além dos literais numéricos incorporados, mostrados na Tabela 4-2, o Python fornece um conjunto de ferramentas para processar objetos de tipo numérico:

Operadores de expressão

`+, *, >>, **` etc.

Funções matemáticas internas

`pow, abs` etc.

Módulos utilitários

`random, math` etc.

* Isto é, a implementação de CPython padrão. Na implementação de Jython, baseada em Java, os tipos do Python são, na verdade, classes Java.

Vamos conhecer todos eles, à medida que prosseguirmos. Finalmente, se você precisar fazer tratamento numérico sério, uma extensão opcional do Python, chamada *NumPy* (Numeric Python – Python numérico) fornece ferramentas de programação numérica avançadas, tal como um tipo de dados matriz e bibliotecas de computação sofisticadas. Grupos de programação científica avançada, em lugares como Lawrence Livermore e NASA, usam Python com NumPy para implementar os tipos de tarefas que anteriormente escreviam em C++ ou FORTRAN.

Como esse é um assunto muito avançado, não falaremos mais nada sobre NumPy neste capítulo. (Veja os exemplos do Capítulo 29.) Você encontrará suporte adicional para programação numérica avançada em Python no site Vaults of Parnassus. Observe também que a NumPy atualmente é uma extensão opcional; ela não vem com o Python e deve ser instalada separadamente.

OPERADORES DE EXPRESSÃO DO PYTHON

Talvez a ferramenta mais básica para processar números seja a *expressão*: uma combinação de números (ou outros objetos) e operadores que calculam um valor quando executados pelo Python. No Python, as expressões são escritas usando a notação matemática normal e símbolos de operadores. Por exemplo, para somar dois números `X` e `Y`, escreva `X+Y`, o que diz ao Python para que aplique o operador `+` nos valores representados por `X` e `Y`. O resultado da expressão é a soma de `X` e `Y`, outro objeto de tipo numérico.

A Tabela 4-3 lista todas as expressões de operadores disponíveis no Python. Muitas são evidentes; por exemplo, os operadores matemáticos normais são aceitos: `+`, `-`, `*`, `/` etc. Alguns você conhece, se já usou C: `%` calcula o resto de uma divisão, `<<` realiza um deslocamento para a esquerda em nível de bit, `&` calcula o resultado de uma função lógica etc. Outros são mais específicos do Python e nem todos têm natureza numérica: o operador `is` testa a igualdade da identidade do objeto (isto é, o endereço), `lambda` cria funções sem nome etc. Mais informações sobre algumas delas, posteriormente.

Operadores misturados: precedência de operador

Assim como na maioria das linguagens, as expressões mais complexas são escritas enfileirando-se as expressões de operador da Tabela 4-3. Por exemplo, a soma de duas multiplicações poderia ser escrita como uma mistura de variáveis e operadores:

```
A * B + C * D
```

Então, como o Python sabe em qual operador deve atuar primeiro? A solução disso está na *precedência do operador*. Quando você escreve uma expressão com mais de um operador, o Python agrupa suas partes de acordo com o que são chamadas de regras de precedência, e esse agrupamento determina a ordem na qual as partes da expressão são calculadas. Na Tabela 4-3, os operadores da parte inferior têm precedência mais alta e, assim, se ligam mais fortemente nas expressões mistas.

Por exemplo, se você escrever `X + Y * Z`, o Python avaliará a multiplicação primeiro (`Y * Z`) e, depois, somará o resultado a `X`, pois `*` tem precedência mais alta (está mais abaixo na tabela) do que `+`. Analogamente, no exemplo original desta seção, as duas multiplicações (`A * B` e `C * D`) acontecerão antes que seus resultados sejam somados.

Tabela 4-3 Operadores de expressão do Python e sua precedência

Operadores	Descrição
`lambda args: expressão`	Geração de função anônima
`x or y`	Função lógica OU (y é avaliado somente se x é falso)
`x and y`	Função lógica E (y é avaliado somente se x for verdadeiro)
`not x`	Negação lógica
`x < y, x <= y, x > y, x >= y,` `x == y, x <> y, x != y,` `x is y, x is not y,` `x in y, x not in y`	Operadores de comparação, operadores de igualdade de valor, testes de identidade de objeto e participação como membro de seqüência
`x | y`	Função lógica OU em nível de bit
`x ^ y`	Função lógica OU EXCLUSIVO em nível de bit
`x & y`	Função lógica E em nível de bit
`x << y, x >> y`	Deslocamento de x à esquerda ou à direita por y bits
`-x + y, x - y`	Adição/concatenação, subtração
`x * y, x % y, x / y, x // y`	Multiplicação/repetição, resto/formato, divisão[a]
`-x, +x, -x, x ** y`	Negação unária, identidade, complemento em nível de bit, potência binária
`x[i], x[i:j], x.attr, x(...)`	Indexação, qualificação, chamadas de função
`(...), [...], {...}, '...'`	Tupla, lista[b], dicionário, conversão para string[c]

[a] A divisão na base (X // Y), novidade na versão 2.2, sempre trunca os restos fracionários. Isso está melhor descrito na seção "Divisão: clássica, na base e real".

[b] A partir do Python 2.0, a sintaxe de lista ([...]) pode representar um literal de lista ou uma expressão de compreensão de lista. Esta última é uma inclusão mais recente do Python, que executa um loop implícito e reúne os resultados da expressão em uma nova lista. Como freqüentemente as abrangências de listas são melhor entendidas em conjunto com funções, elas foram deixadas para o Capítulo 14.

[c] A conversão de objetos em suas strings de impressão também pode ser realizada com as funções internas `str` e `repr`, mais legíveis, as quais estão descritas na seção "Representação numérica".

Sub-expressões agrupadas com parênteses

Você pode esquecer a precedência completamente, se tomar o cuidado de agrupar as partes das expressões com parênteses. Quando coloca sub-expressões entre parênteses, você anula as regras de precedência do Python. O programa sempre avalia as expressões entre parênteses primeiro, antes de usar seus resultados nas expressões englobadas.

Por exemplo, em vez de codificar `X + Y * Z`, escreva uma das expressões a seguir para obrigar o Python a avaliar a expressão na ordem desejada:

```
(X + Y) * Z
X + (Y * Z)
```

No primeiro caso, `+` é aplicado a `X` e `Y` primeiro, pois está dentro dos parênteses. No segundo caso, `*` é executado primeiro (exatamente como se não houvesse parênteses). Falando de modo geral, adicionar parênteses em expressões grandes é uma excelente idéia; isso não apenas impõe a ordem de avaliação desejada, mas também auxilia na legibilidade.

Tipos misturados: convertidos para cima

Além de misturar operadores em expressões, você também pode misturar tipos numéricos. Por exemplo, você pode somar um valor inteiro a um número em ponto flutuante:

```
40 + 3.14
```

Mas isso leva a outra questão: qual é o tipo de resultado – inteiro ou em ponto flutuante? A resposta é simples, especialmente se você já usou praticamente qualquer outra linguagem: nas expressões de tipo misto, o Python primeiro converte os operandos *para cima*, para o tipo de operando mais complicado, e depois efetua o cálculo matemático em operandos de mesmo tipo. Se você já usou a linguagem C, achará isso semelhante às conversões de tipo dessa linguagem.

O Python classifica a complexidade dos tipos numéricos, como segue: os inteiros são mais simples do que os inteiros longos, que são mais simples do que os números de ponto flutuante, os quais são mais simples do que os números complexos. Então, quando um valor inteiro é misturado com um valor em ponto flutuante, como no exemplo, o inteiro é primeiro convertido para um valor de ponto flutuante e a matemática de números de ponto flutuante gera o resultado em ponto flutuante. Analogamente, qualquer expressão de tipo misto onde um operando é um número complexo, resulta na conversão do outro operando para um número complexo, o que gera um resultado complexo.

Conforme você verá posteriormente nesta seção, a partir do Python 2.2, a linguagem também converte automaticamente inteiros normais em inteiros longos, quando seus valores são grandes demais para caber em um inteiro normal. Lembre-se também de que todas essas conversões de tipo misto só se aplicam na mistura de tipos *numéricos* em torno de um operador ou de uma comparação (por exemplo, um inteiro e um número em ponto flutuante). Em geral, o Python não faz conversão além dos limites de outro tipo. Somar uma string a um inteiro, por exemplo, resulta em um erro, a não ser que você converta um ou outro manualmente; esteja atento para um exemplo quando conhecermos as strings no Capítulo 5.

Prévia: sobrecarga de operador

Embora estejamos focando agora nos números internos, lembre-se de que todos os operadores do Python podem ser sobrepostos (isto é, implementados) por classes do Python e por tipos de extensão da linguagem C, para trabalhar nos objetos criados por você. Por exemplo, você verá posteriormente que objetos desevolvidos com classes podem ser somados com expressões `+`, indexados com expressões `[i]` etc.

Além disso, alguns operadores já são sobrepostos pelo próprio Python; eles executam ações diferentes, dependendo do tipo de objeto interno que esteja sendo processado. Por exemplo, o operador `+` efetua a adição quando aplicado a números, mas realiza uma concatenação quando aplicado a objetos de uma seqüência, como strings e listas.*

NÚMEROS EM AÇÃO

Provavelmente, a melhor maneira de entender os objetos e expressões numéricas é vê-los em ação. Então, inicie a linha de comando interativa e digite algumas operações básicas, porém ilustrativas.

Operações básicas e variáveis

Antes de tudo, vamos efetuar algumas operações matemáticas básicas. Na interação a seguir, primeiro atribuímos valores inteiros a duas *variáveis* (`a` e `b`), para que possamos usá-las poste-

* Normalmente, isso se chama *polimorfismo* – o significado de uma operação depende do tipo dos objetos que estão participando dessa operação. Vamos rever esse termo quando explorarmos as funções, no Capítulo 12, pois lá ele se torna um recurso muito mais óbvio.

riormente em uma expressão maior. As variáveis são simplesmente nomes – criados por você ou pelo Python – usados para monitorar informações em seu programa. Vamos falar sobre isso posteriormente, mas no Python:

- As variáveis são criadas na primeira vez que recebem um valor.
- As variáveis são substituídas por seus valores, quando usadas em expressões.
- As variáveis devem receber atribuições antes que possam ser usadas em expressões.
- As variáveis se referem a objetos e nunca são declaradas antecipadamente.

Em outras palavras, as atribuições fazem essas variáveis existirem automaticamente.

```
% python
>>> a = 3                  # Nome criado
>>> b = 4
```

Também usamos um *comentário* aqui. No código em Python, o texto após uma marca # e continuando até o final da linha, é considerado como comentário e é ignorado pela linguagem. Os comentários são lugares para se escrever documentação legível para seres humanos, em seu código. Como o código digitado interativamente é temporário, normalmente você não escreverá comentários lá, mas eles foram inseridos nos exemplos para ajudar a explicar o código.* Na próxima parte deste livro, conheceremos um recurso relacionado – as strings de documentação – que anexa o texto de seus comentários em objetos.

Agora, vamos usar os objetos de valor inteiro em expressões. Neste ponto, `a` e `b` ainda são `3` e `4` respectivamente; variáveis como essas são substituídas por seus valores quando usadas dentro de uma expressão e os resultados da expressão são ecoados ao se trabalhar interativamente:

```
>>> a + 1, a - 1            # Adição (3+1), subtração (3-1)
(4, 2)

>>> b * 3, b / 2            # Multiplicação (4*3), divisão (4/2)
(12, 2)

>>> a % 2, b ** 2           # Módulo (resto), potenciação
(1, 16)

>>> 2 + 4.0, 2.0 ** b       # Conversões de tipo misto
(6.0, 16.0)
```

Tecnicamente, os resultados que estão sendo ecoados aqui são *tuplas* de dois valores, pois as linhas digitadas no prompt contêm duas expressões separadas por vírgulas. É por isso que o resultado é exibido entre parênteses (mais informações sobre tuplas, posteriormente). Note que as expressões funcionam porque as variáveis `a` e `b` que estão dentro delas receberam valores; se você usar uma variável diferente que nunca recebeu nenhuma atribuição, o Python relatará um erro, em vez de inserir algum valor padrão:

```
>>> c * 2
Traceback (most recent call last):
    File "<stdin>", line 1, in?
NameError: name 'c' is not defined
```

Você não precisa declarar variáveis previamente no Python, mas elas devem receber atribuição pelo menos uma vez, antes de poderem ser usadas. Aqui estão duas expressões ligeira-

* Se você estiver acompanhando no computador, não precisa digitar nenhum texto de comentário, desde # até o final da linha; os comentários são simplesmente ignorados pelo Python e não são uma parte obrigatória das instruções que executamos.

mente maiores para ilustrar o agrupamento de operadores e mais informações a respeito de conversões:

```
>>> b / 2 + a                # É o mesmo que ((4 / 2) + 3)
5
>>> print b / (2.0 + 1)      # É o mesmo que (4 / (2.0 + 3))
0.8
```

Na primeira expressão, não existem parênteses; portanto, o Python agrupa automaticamente os componentes, de acordo com suas regras de precedência – como / está mais embaixo na Tabela 4-3 do que +, ele se liga mais fortemente e, assim, é avaliado primeiro. O resultado é como se a expressão tivesse parênteses, como mostrado no comentário à direita do código. Note também que todos os números são inteiros na primeira expressão; por isso, o Python efetua divisão e adição de inteiros.

Na segunda expressão, são colocados parênteses em torno da parte + para obrigar o Python a avaliá-lo primeiro (isto é, antes de `/`). Também transformamos um dos operandos para ponto flutuante, adicionando um ponto decimal: `2.0`. Por causa dos tipos mistos, o Python converte o valor inteiro referenciado em um valor de ponto flutuante (`3.0`), antes de efetuar `+`. Ele também converte `b` para um valor de ponto flutuante (`4.0`) e efetua uma divisão de ponto flutuante; (`4.0/5.0`) gera um resultado de ponto flutuante igual a `0.8`. Se todos os números dessa expressão fossem inteiros, ele efetuaria uma divisão de inteiros (`4/5` e o resultado seria o inteiro truncado 0 (no Python 2.2, pelo menos – veja a discussão sobre a divisão real, mais adiante).

Representação numérica

A propósito, note que usamos uma instrução `print` no segundo exemplo; sem essa instrução, você verá algo que, a primeira vista, pode parecer um pouco estranho:

```
>>> b / (2.0 + a)            # Saída de eco automática: mais dígitos
0.80000000000000004

>>> print b / (2.0 + a)      # print arredonda os dígitos.
0.8
```

A história inteira por trás disso tem a ver com as limitações do hardware de ponto flutuante e sua incapacidade de representar alguns valores com precisão. Contudo, como a arquitetura de computador está fora dos objetivos deste livro, vamos simplificar dizendo que todos os dígitos na primeira saída estão realmente lá, no hardware de ponto flutuante do seu computador; você apenas, normalmente, não está acostumado a vê-los. Estamos usando esse exemplo para demonstrar a diferença na formatação da saída – o eco do resultado automático do prompt interativo mostra mais dígitos do que a instrução `print`. Se você não quiser todos os dígitos, escreva `print`.

Note que nem todos os valores têm tantos dígitos para exibir:

```
>>> 1 / 2.0
0.5
```

E existem mais maneiras de exibir os bits de um número dentro de seu computador do que as instruções print e os ecos automáticos:

```
>>> num = 1 / 3.0
>>> num                      # Ecoa
0.33333333333333331
```

```
>>> print num                 # Print arredonda
0.333333333333

>>> "%e" % num                # Formatação de string
'3.333333e-001'

>>> "%2.2f" % num             # Formatação de string
'0.33'
```

Os dois últimos exemplos empregam *formatação de string* – uma expressão que possibilita flexibilidade na formatação, algo explorado no próximo capítulo sobre strings.

Formatos de exibição de str e repr

Tecnicamente, a diferença entre os ecos interativos padrão e as instruções print corresponde à diferença entre as funções internas `repr` e `str`:

```
>>> repr(num)                 # Usada por ecos: como forma de código
'0.33333333333333331'
>>> str(num)                  # Usada por print: forma amigável para o usuário
'0.333333333333'
```

Ambas convertem objetos arbitrários em sua representação de string: `repr` (e o prompt interativo) produz resultados que têm aparência de código; `str` (e a instrução `print`) converte para um formato normalmente mais amigável para o usuário. Essa noção virá à tona novamente, quando estudarmos as strings. Mais informações sobre esses detalhes internos em geral aparecerão posteriormente no livro.

Divisão: clássica, na base e real

Agora que você já viu como a divisão funciona, deve saber que ela está programada para mudar em uma futura versão do Python (atualmente, na 3.0, programada para aparecer alguns anos depois desta edição do livro ser lançada). No Python 2.3, as coisas funcionam como acabamos de descrever, mas na verdade existem dois operadores de divisão diferentes, um dos quais mudará:

`X / Y`

Divisão *clássica*. No Python 2.3 e anteriores, esse operador trunca os resultados de inteiros e mantém o resto para números de ponto flutuante, conforme descrito aqui. Esse operador será alterado para divisão *real* – sempre mantendo o resto, independente dos tipos – em uma versão futura do Python (3.0).

`X // Y`

Divisão *na base*. Adicionado no Python 2.2, esse operador sempre trunca os restos fracionários em suas bases, independente dos tipos.

A divisão na base foi adicionada para tratar do fato de que o resultado do modelo da divisão clássica atual é dependente dos tipos de operando e, assim, às vezes pode ser difícil de prever em uma linguagem de tipagem dinâmica, como o Python.

Devido aos possíveis problemas de compatibilidade com versões anteriores, isso está em estado de mudança atualmente. Na versão 2.3, a divisão / funciona conforme descrito, por

padrão, e a divisão na base // foi adicionada para truncar o resto dos resultados em suas bases, independente dos tipos:

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2),
(2, 2.5, -2.5, -3)

>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2),
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0),
(3, 3.0, 3, 3.0)
```

Em uma versão futura do Python, a divisão / provavelmente será alterada para retornar um resultado de divisão real, que sempre mantém os restos, mesmo para inteiros – por exemplo, `1/2` será `0.5` e não `0`, e `1//2` ainda será `0`.

Até que essa alteração seja completamente incorporada, você pode ver como / provavelmente funcionará no futuro, usando uma importação especial desta forma: `from __future__ import division`. Isso transforma o operador / em uma divisão real (mantendo os restos), mas deixa // como está. Aqui está como / finalmente se comportará:

```
>>> from __future__ import division

>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2),
(2.5, 2.5, -2.5, -2.5)

>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2),
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0),
(3.0, 3.0, 3, 3.0)
```

Fique atento para um exemplo simples de loop `while` de números primos no Capítulo 10, e para um exercício correspondente, no final da Parte IV, que ilustra o tipo de código que pode ser afetado por essa alteração de /. Em geral, qualquer código que dependa de / para truncar um resultado inteiro pode ser afetado (em vez disso, use o novo operador //). Quando escrevemos isto, a alteração estava programada para ocorrer no Python 3.0, mas experimente essas expressões em sua versão para ver qual comportamento se aplica. Além disso, fique atento para mais informações sobre o comando `from` especial usado aqui, no Capítulo 18.

Operações em nível de bit

Além das operações numéricas normais (adição, subtração etc.), o Python suporta a maioria das expressões numéricas disponíveis na linguagem C. Por exemplo, aqui está o funcionamento de operações de deslocamento em nível de bit e booleanas:

```
>>> x = 1          # 0001
>>> x << 2         # Desloca 2 bits para a esquerda: 0100
4
>>> x | 2          # Função lógica OU em nível de bit: 0011
3
>>> x & 1          # Função lógica E em nível de bit: 0001
1
```

Na primeira expressão, um valor `1` em binário (na base `2, 0001`) é deslocado duas casas para a esquerda para criar um valor `4` em binário (`0100`). As duas últimas operações executam uma função lógica binária ou (`0001|0010 = 0011`) e uma função lógica binária e (`0001&0001 = 0001`).

Tais operações de mascaramento de bits nos permitem codificar vários flags e outros valores dentro de um único inteiro.

Não entraremos em muitos detalhes sobre "brincadeiras com bits" aqui. Caso você precise, elas são suportadas, mas saiba que, freqüentemente, isso não é tão importante em uma linguagem de alto nível, como o Python, quanto é em uma linguagem de baixo nível, como o C. Como regra geral, se você quiser mexer com bits no Python, deve pensar em qual linguagem está escrevendo realmente. Em geral, freqüentemente existem maneiras melhores de codificar informações no Python do que as strings de bits.*

Inteiros longos

Agora, vamos ver algo mais exótico. Aqui está um exemplo de inteiros longos em ação. Quando um literal inteiro termina com a letra `L` (ou `l` minúsculo), o Python cria um *inteiro longo*. No Python, um inteiro longo pode ser arbitrariamente grande – ele pode ter tantos dígitos quanto você tiver de espaço na memória:

```
>>> 9999999999999999999999999999999999999L + 1
10000000000000000000000000000000000000L
```

O `L` no final da string de dígitos diz ao Python para que crie um objeto de tipo inteiro longo com precisão ilimitada. A partir do Python 2.2, até a letra `L` é opcional, de modo geral – o Python converte automaticamente os inteiros normais em inteiros longos, quando eles estouram a precisão dos inteiros normais (normalmente, 32 bits):

```
>>> 9999999999999999999999999999999999999 + 1
10000000000000000000000000000000000000L
```

Os inteiros longos são uma ferramenta interna conveniente. Por exemplo, você pode usá-los para contar dívidas públicas em centavos, diretamente no Python (caso tenha essa propensão e haja memória suficiente em seu computador). Eles também são o motivo pelo qual pudemos elevar 2 a potências grandes nos exemplos do Capítulo 3:

```
>>> 2L ** 200
1606938044258990275541962092341162602522202993782792835301376L
>>>
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376L
```

Como o Python precisa realizar trabalho extra para suportar sua precisão estendida, normalmente a matemática de inteiros longos é significativamente mais lenta do que a matemática de inteiros normais (que normalmente é mapeada diretamente no hardware). Se você precisar de precisão, ela está incorporada para ser usada, mas o desempenho é prejudicado.

Uma nota sobre mudança na versão: antes do Python 2.2, os inteiros não eram convertidos automaticamente em inteiros longos em caso de overflow; portanto, você tinha que realmente usar a letra `L` para obter a precisão estendida:

```
>>> 9999999999999999999999999999999999999 + 1      # Antes da 2.2
OverFlowError: integer literal too large

>>> 9999999999999999999999999999999999999L + 1     # Antes da 2.2
10000000000000000000000000000000000000L
```

* Assim como na maioria das regras, existem exceções. Por exemplo, se você mexer com bibliotecas C que esperam a passagem de strings de bits, isso não se aplica.

Na versão 2.2, a letra L é opcional, de modo geral. No futuro, é possível que usar a letra L possa gerar um alerta. Por isso, provavelmente é melhor você deixar o Python fazer a conversão automaticamente, quando for necessário, e omitir a letra L.

Números complexos

Os números complexos são um tipo de objeto básico diferente no Python. Se você sabe o que eles são, então sabe por que são úteis; se não, considere esta seção uma leitura opcional. Os números complexos são representados como dois números de ponto flutuante – as partes real e imaginária – e são escritos adicionando um sufixo j ou J na parte imaginária. Também podemos escrever números complexos com uma parte real diferente de zero, adicionando as duas partes com um +. Por exemplo, o número complexo com uma parte real igual a 2 e uma parte imaginária igual a -3 é escrito como 2 + -3j. Aqui estão alguns exemplos de matemática de números complexos em funcionamento:

```
>>> 1j * 1J
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2+1j)*3
(6+3j)
```

Os números complexos também nos permitem extrair suas partes como atributos, suportam todas as expressões matemáticas normais e podem ser processados com ferramentas do módulo cmath padrão (a versão complexa do módulo math padrão). Normalmente, os números complexos são empregados em programas voltados para engenharia. Como eles são uma ferramenta avançada, consulte o manual de referência da linguagem Python para ver os detalhes adicionais.

Notação hexadecimal e octal

Conforme mencionado no início desta seção, os inteiros do Python podem ser escritos na notação hexadecimal (base 16) e octal (base 8), além da escrita decimal normal, de base 10:

- Os literais em octal têm um 0 inicial, seguido de uma string de dígitos 0-7 em octal, cada um dos quais representando 3 bits.
- Os literais em hexadecimal têm um 0x ou 0X inicial, seguido de uma string de dígitos 0-9 e letras maiúsculas ou minúsculas de A-F em hexadecimal, cada um dos quais representando 4 bits.

Lembre-se de que isso é simplesmente uma sintaxe alternativa para especificar o valor de um objeto inteiro. Por exemplo, os literais em octal e em hexadecimal a seguir produzem inteiros normais, com os valores especificados:

```
>>> 01, 010, 0100                    # Literais em octal
(1, 8, 64)
>>> 0x01, 0x10, 0xFF                 # Literais em hexadecimal
(1, 16, 255)
```

Aqui, o valor em octal 0100 é 64 em decimal, e o hexadecimal 0xFF é 255 em decimal. Por padrão, o Python imprime em decimal, mas fornece funções internas que permitem converter inteiros em suas strings de dígitos em octal e hexadecimal:

```
>>> oct(64), hex(64), hex(255)
('0100', '0x40', 0xff')
```

A função `oct` converte de decimal para octal e `hex` converte para hexadecimal. Para fazer o oposto, a função interna `int` converte uma string de dígitos em um inteiro; um segundo argumento opcional permite especificar a base numérica:

```
>>> int('0100'), int('0100', 8), int('0x40', 16)
(100, 64, 64)
```

A função `eval`, que você conhecerá posteriormente neste livro, trata as strings como se fossem código em Python. Portanto, ela tem um efeito semelhante (mas normalmente é executada mais lentamente – na verdade, ela compila e executa a string como um trecho de programa):

```
>>> eval('100'), eval('0100'), eval('0x40')
(100, 64, 64)
```

Finalmente, você também pode converter inteiros em strings, em octal e hexadecimal, com uma expressão de formatação de string:

```
>>> "%0 %x %X" % (64, 64, 255)
'100 40 FF'
```

Isso será abordado no Capítulo 5.

Um alerta antes de prosseguirmos: cuidado para não iniciar uma string com um zero no Python, a não ser que realmente queira escrever um valor em octal. O Python a tratará como sendo de base 8, o que pode não funcionar como você esperaria – 010 é sempre 8 em decimal e não o valor decimal 10 (apesar do que você poderia pensar!).

Outras ferramentas numéricas

O Python também fornece funções e módulos internos para processamento numérico. Aqui estão exemplos do módulo interno `math` e de algumas funções internas em funcionamento.

```
>>> import math
>>> math.pi, math.e
(3.1415926535897931, 2.7182818284590451)

>>> math.sin(2 * math.pi / 180)
0.034899496702500969

>>> abs(-42), 2**4, pow(2, 4)
(42, 16, 16)

>>> int(2.567), round(2.567), round(2.567, 2)
(2, 3.0, 2.5699999999999998)
```

O módulo `math` contém a maioria das ferramentas da biblioteca matemática da linguagem C. Conforme descrito anteriormente, a última saída aqui será apenas `2.57`, se escrevermos `print`.

Note que os módulos internos, como o `math`, devem ser importados, mas as funções internas, como as `abs`, estão sempre disponíveis sem importações. Em outras palavras, os módulos são componentes externos, mas as funções internas ficam em um espaço de nomes implícito, o qual o Python pesquisa automaticamente para encontrar nomes usados em seu programa. Esse espaço de nomes corresponde ao módulo chamado `__builtin__`. Há muito mais informações sobre solução de espaço de nomes na Parte IV, Funções. Por enquanto, quando você ler "módulo", pense em "importação".

O ENTREATO DA TIPAGEM DINÂMICA

Se você tem conhecimento prévio de linguagens compiladas ou de tipagem estática, como C, C++ ou Java, pode estar confuso neste ponto. Até aqui, estivemos usando variáveis sem declarar seus tipos – e isso funciona de algum modo. Quando digitamos `a = 3` em uma sessão interativa ou em um arquivo de programa, como o Python sabe que deve representar um inteiro? Quanto a isso, como o Python até mesmo sabe o que é `a`?

Quando você começa a fazer essas perguntas, já entrou no domínio do modelo de *tipagem dinâmica* do Python. Nessa linguagem, os tipos são determinados automaticamente, em tempo de execução, e não em resposta às declarações presentes em seu código. Para você, isso significa que nunca é preciso declarar variáveis antecipadamente e talvez esse seja um conceito mais simples, se não tiver programado em outras linguagens. Contudo, como esse provavelmente é o conceito mais fundamental da linguagem, vamos explorá-lo em detalhes aqui.

Como as atribuições funcionam

Você vai notar que, quando escrevemos `a = 3`, isso funciona, mesmo que nunca tenhamos dito ao Python para que utilize o nome `a` como variável. Além disso, a atribuição de `3` a `a` também parece funcionar, mesmo que não tenhamos dito ao Python que deve significar um objeto de tipo inteiro. Na linguagem Python, tudo isso resulta de uma maneira muito natural, como segue:

Criação

Uma variável, como `a`, é criada ao receber um valor pela primeira vez em seu código. As atribuições futuras alteram o nome já criado de modo a ter um novo valor. Tecnicamente, o Python detecta alguns nomes antes que seu código seja executado; mas, conceitualmente, você pode considerar como se as atribuições produzissem as variáveis.

Tipos

Uma variável, como `a`, nunca tem qualquer informação de tipo nem restrição associada. Em vez disso, a noção de tipo está presente em objetos e não em nomes. As variáveis sempre se referem simplesmente a um objeto em particular, em dado momento específico.

Uso

Quando uma variável aparece em uma expressão, ela é substituída imediatamente pelo objeto a que se refere correntemente, qualquer que seja ele. Além disso, todas as variáveis devem receber atribuição explicitamente, antes de poderem ser usadas; o uso de variáveis que não receberam atribuição resulta em um erro.

Esse modelo é completamente diferente das linguagens tradicionais e é responsável por grande parte da característica concisa e da flexibilidade do Python. Quando você é iniciante, a tipagem dinâmica normalmente é mais fácil de entender, se mantiver clara a distinção entre nomes e objetos. Por exemplo, quando escrevemos:

```
>>> a = 3
```

Pelo menos conceitualmente, o Python executará três passos distintos para atender o pedido, os quais refletem a operação de todas as atribuições na linguagem Python:

1. Criar um objeto para representar o valor `3`.
2. Criar a variável `a`, caso ela ainda não exista.
3. Vincular a variável `a` ao novo objeto `3`.

O resultado será uma estrutura dentro do Python, semelhante à Figura 4-1. Conforme está esboçado, as variáveis e os objetos são armazenados em partes diferentes da memória e associados por vínculos – mostrados como uma seta na figura. As variáveis sempre se vinculam a objetos (e nunca a outras variáveis), mas os objetos maiores podem se vincular a outros objetos.

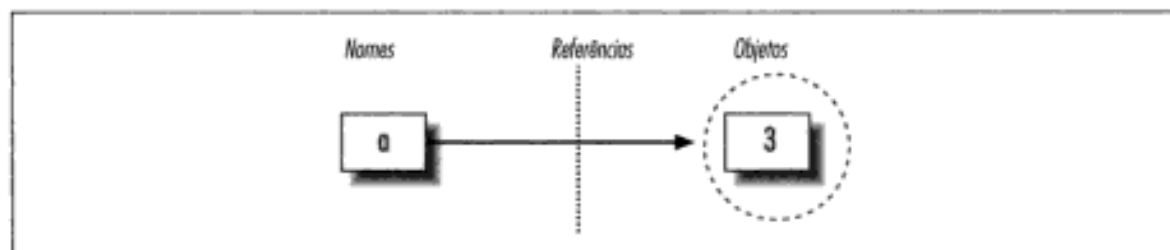



Figura 4-1 Nomes e objetos, após a = 3.

Esses vínculos das variáveis para os objetos são chamados de *referências* no Python – uma espécie de associação.* Quando as variáveis são usadas (isto é, referenciadas) posteriormente, os vínculos da variável para o objeto são seguidos automaticamente pelo Python. Tudo isso é mais simples do que a terminologia pode sugerir. Em termos concretos:

- As *variáveis* são simplesmente entradas em uma tabela de pesquisa, com espaço para vínculo para um objeto.
- Os *objetos* são apenas partes de memória alocada, com espaço suficiente para representar o valor que denotam e informações de `tag` de tipo.

Pelo menos conceitualmente, sempre que você gera um novo valor em seu script, o Python cria um novo objeto (isto é, um trecho de memória) para representar esse valor. O Python coloca em cache e reutiliza certos tipos de objetos inalteráveis, como inteiros pequenos e strings, como forma de otimização (cada zero não é realmente um novo trecho de memória); mas isso funciona como se cada valor fosse um objeto distinto. Vamos rever esse conceito quando conhecermos as comparações `==` e `is`, na seção "Comparações, igualdade e verdade", no Capítulo 7.

Vamos prolongar a sessão e ver o que acontece com seus nomes e objetos:

```
>>> a = 3
>>> b = a
```

Após digitarmos essas duas instruções, geramos o cenário capturado na Figura 4-2. Como antes, a segunda linha faz o Python criar a variável `b`; aqui, a variável `a` está sendo usada e não está recebendo uma atribuição, de modo que ela é substituída pelo objeto que referencia (`3`); e `b` faz referência a esse objeto. O efeito global é que as variáveis `a` e `b` acabam referenciando o mesmo objeto (isto é, apontando para o mesmo trecho de memória). Isso é chamado de *referência compartilhada* no Python – vários nomes referenciando o mesmo objeto.

* Os leitores com experiência em C podem achar as referências do Python semelhantes aos ponteiros (para endereços de memória) da linguagem C. Na verdade, as referências são implementadas como ponteiros e freqüentemente têm as mesmas funções, especialmente com objetos que podem ser alterados no local (mais informações sobre isso posteriormente). Entretanto, como as referências são sempre desfeitas automaticamente quando usadas, você nunca pode fazer nada de útil com uma referência sozinha; esse é um recurso que elimina uma enorme gama de erros da linguagem C. Mas você pode considerar as referências do Python como ponteiros "void" da linguagem C, os quais, quando usados, são seguidos automaticamente.

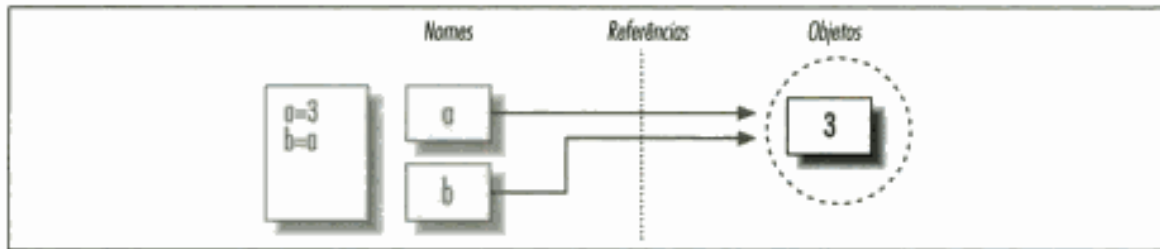



Figura 4-2 Nomes e objetos, após b = a.

Em seguida, suponha que prolonguemos a sessão com mais uma instrução:

```
>>> a = 3
>>> b = a
>>> a = 'spam'
```

Assim como em todas as atribuições do Python, isso simplesmente cria um novo objeto para representar o valor da string "spam" e configura a para referenciar esse novo objeto. Entretanto, isso não altera o valor de `b`; `b` ainda se refere ao objeto original, o valor inteiro `3`. A estrutura de referência resultante aparece na Figura 4-3.

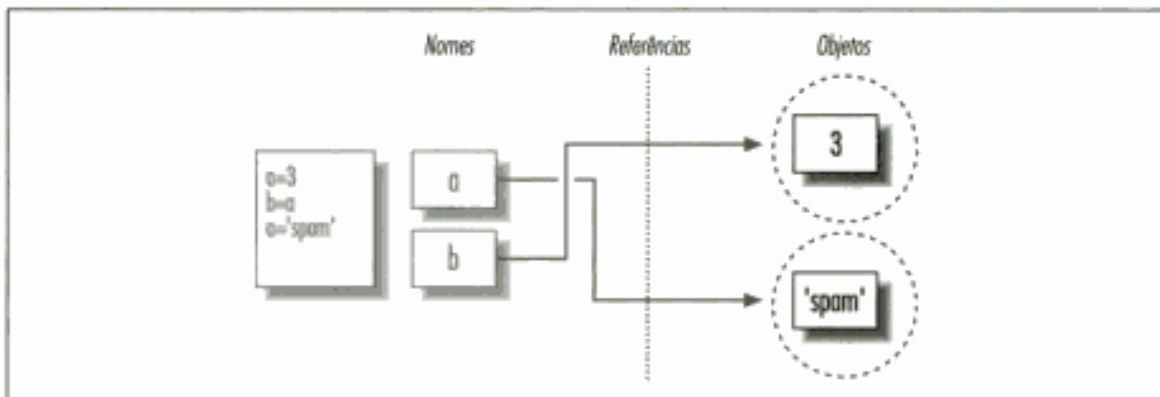



Figura 4-3 Nomes e objetos, após a = 'spam'.

O mesmo tipo de coisa aconteceria se, em vez disso, alterássemos `b` para "spam" – a atribuição alteraria apenas `b` e não `a`. Esse exemplo tende a parecer particularmente estranho para ex-programadores de C – parece que, escrevendo-se `a = 'spam'`, o *tipo* de `a` mudou de inteiro para string. Mas na realidade não é assim. No Python, as coisas funcionam de modo mais simples: os tipos pertencem aos objetos e não aos nomes. Simplesmente alteramos `a` para referenciar um objeto diferente.

Esse comportamento também ocorre se não houver nenhuma diferença de tipo. Por exemplo, considere estas três instruções:

```
>>> a = 3
>>> b = a
>>> a = 5
```

Nessa seqüência, os mesmos eventos ocorrem: o Python faz a variável `a` referenciar o objeto `3` e faz `b` referenciar o mesmo objeto que `a`, como na Figura 4-2. Como antes, a última atribuição apenas configura `a` com um objeto completamente diferente, o valor inteiro `5`. Ela não altera `b` como um efeito colateral. Na verdade, não há nenhuma maneira de sobrescrever o valor do objeto `3` (os inteiros nunca podem ser alterados no local – uma propriedade chamada

imutabilidade). Ao contrário de algumas linguagens, as variáveis do Python são sempre ponteiros para objetos e não rótulos de áreas de memória alteráveis.

Referências e objetos alteráveis

Contudo, conforme você verá posteriormente nos capítulos desta parte, existem objetos e operações que realizam alterações em objetos no local. Por exemplo, a atribuição de deslocamentos em listas altera realmente objeto de tipo lista (no local) em si, em vez de gerar um objeto novo. Para objetos que suportam tais alterações no local, você precisa saber mais sobre as referências compartilhadas, pois a alteração de um nome pode afetar outros. Por exemplo, os objetos tipo lista suportam atribuição no local para posições:

```
>>> L1 = [2,3,4]
>>> L2 = L1
```

Conforme observado no início deste capítulo, as listas são simplesmente coleções de outros objetos, escritos entre colchetes. Aqui, `L1` é uma lista contendo os objetos `2, 3` e `4`. Os itens dentro de uma lista são acessados por meio de suas posições; `L1[0]` refere-se ao objeto `2`, o primeiro item da lista `L1`.

As listas também são objetos propriamente ditos, assim como os inteiros e as strings. Após a execução das duas atribuições anteriores, `L1` e `L2` referenciam o mesmo objeto, exatamente como no exemplo anterior (veja a Figura 4-2). Também como antes, se agora escrevermos:

```
>>> L1 = 24
```

`L1` será simplesmente configurada com um objeto diferente; `L2` ainda é a lista original. Entretanto, se em vez disso alterarmos ligeiramente a sintaxe da instrução, ela terá um efeito radicalmente diferente:

```
>>> L1[0] = 24
>>> L2
[24, 3, 4]
```

Aqui, alteramos um componente do *objeto* que `L1` referencia, em vez de alterar `L1` em si. Esse tipo de alteração sobrescreve parte do objeto de tipo lista no local. O resultado final é que o efeito também aparece em `L2`, pois ele compartilha o mesmo objeto que `L1`.

Normalmente, é isso que você deseja, mas é preciso saber como funciona para que seja algo esperado. Esse também é apenas o padrão; se você não quiser esse comportamento, pode pedir para que o Python copie os objetos, em vez de fazer referências. Vamos explorar as listas com mais profundidade e rever o conceito de referências compartilhadas e cópias nos Capítulos 6 e 7.*

Referências e coleta de lixo

Quando os nomes referenciam objetos novos, o Python também recupera o objeto antigo, caso ele não seja referenciado por nenhum outro nome (ou objeto). Essa recuperação automática do espaço do objeto é conhecida como *coleta de lixo*. Isso significa que você pode usar os objetos livremente, sem jamais precisar liberar espaço em seu script. Na prática, isso elimina

* Os objetos que podem ser alterados no local são conhecidos como *mutáveis* – as listas e os dicionários são componentes internos mutáveis e, portanto, suscetíveis aos efeitos colaterais da alteração no local.

uma quantidade substancial de código de contabilidade, em comparação com as linguagens de nível mais baixo, como C e C++.

Para ilustrar, considere o código a seguir, que configura o nome x com um objeto diferente em cada atribuição. Primeiro, observe como o nome x é configurado com um *tipo* diferente de objeto a cada vez. É como se o tipo de x estivesse mudando com o passar do tempo; mas isso não acontece realmente – no Python, os tipos pertencem aos objetos e não aos nomes. Como os nomes são apenas referências genéricas para os objetos, esse tipo de código funciona naturalmente:

```
>>> x = 42
>>> x = 'shrubberry'                    # Recupera 42 agora (?)
>>> x = 3.1415                          # Recupera 'shrubberry' agora (?)
>>> x = [1,2,3]                         # Recupera 3.1415 agora (?)
```

Em segundo lugar, observe que as referências para os *objetos* são descartadas no processo. Sempre que x recebe um novo objeto, o Python recupera o objeto anterior. Por exemplo, quando x receber a string 'shrubberry', o objeto 42 será recuperado imediatamente, contanto que não seja referenciado em nenhum outro lugar – o espaço do objeto é devolvido automaticamente para o pool de espaço livre, para ser reutilizado por um futuro objeto.

Tecnicamente, para certos tipos, esse comportamento de coleta pode ser mais conceitual do que literal. Como o Python coloca em cache e reutiliza inteiros e strings pequenas, conforme mencionado anteriormente, o objeto 42 provavelmente não será recuperado literalmente; ele permanecerá, para ser usado na próxima vez que você gerar um valor 42 em seu código. Contudo, a maioria dos tipos de objetos são recuperados imediatamente, quando o objeto não é mais referenciado. Para aqueles que não são recuperados, o mecanismo de cache é irrelevante para seu código.

É claro que você não precisa desenhar diagramas de nome/objeto, com círculos e setas, para utilizar o Python. Contudo, quando você está começando a aprender, isso às vezes ajuda a entender alguns casos incomuns, se conseguir acompanhar sua estrutura de referência. Além disso, como *tudo* parece ser atribuições e referências no Python, um entendimento básico desse modelo ajuda em muitos contextos – conforme veremos, isso funciona da mesma forma nas instruções de atribuição, para variáveis de loop for, argumentos de função, importações de módulo e muito mais.

5 Strings

O próximo tipo interno importante no Python é conhecido como *string* – um conjunto ordenado de caracteres, usado para armazenar e representar informações textuais. Da perspectiva funcional, as strings podem ser usadas para representar praticamente tudo que pode ser escrito como texto: símbolos e palavras (seu nome, por exemplo), conteúdo de arquivos de texto carregados na memória, endereços de Internet, programas em Python etc.

Talvez você também já tenha usado strings em outras linguagens; as strings do Python têm a mesma função dos arrays de caracteres de linguagens como C, mas no Python a string é uma ferramenta de nível bem mais alto do que os arrays. Ao contrário da linguagem C, as strings do Python possuem um poderoso conjunto de ferramentas de processamento. Também diferentemente de linguagens como C, o Python não tem nenhum tipo especial para caracteres simples (como `char` da linguagem C), apenas strings de um só caractere.

Rigorosamente falando, as strings do Python são classificadas como *seqüências imutáveis* – significando que elas têm uma ordem (*seqüência*) posicional que vai da esquerda para a direita e não podem ser alteradas no local (*imutáveis*). Na verdade, as strings são as primeiras representantes da classe mais ampla de objetos chamados seqüências. Preste particular atenção nas operações apresentadas aqui, pois elas funcionarão de forma igual em outros tipos de seqüências que veremos posteriormente, como as listas e as tuplas.

A Tabela 5-1 apresenta os literais e operações de string comuns. As strings vazias são escritas com duas aspas sem nada entre elas, e existem várias maneiras de escrever strings. Para processamento, as strings suportam operações de *expressão*, como concatenação (combinação de strings), fracionamento (extração de seções), indexação (busca pelo deslocamento) etc. Além das expressões, o Python também fornece um conjunto de *métodos* que implementam tarefas comuns específicas das strings, assim como *módulos*, que espelham a maioria dos métodos de string.

Os métodos e módulos serão discutidos posteriormente nesta seção. Além do conjunto básico de ferramentas de string, o Python também suporta processamento de string mais avançado, baseado em padrões, com o módulo `re` (expressão regular) da biblioteca padrão, apresentado no Capítulo 27. Esta seção começa com uma visão geral das formas de literal de string e das expressões de string básicas; em, seguida, examinaremos ferramentas mais avançadas, como métodos e formatação de string.

Tabela 5-1 Literais e operações de string comuns

Operação	Interpretação
`s1 = ''`	String vazia
`s2 = "spam's"`	Aspas duplas
`block = """..."""`	Blocos com aspas triplas
`s3 = r'\temp\spam'`	Strings brutas
`s4 = u'spam'`	Strings Unicode
`s1 + s2`	Concatenação,
`s2 * 3`	repetição
`s2[i]`	Índice,
`s2[i:j]`	fracionamento,
`len(s2)`	comprimento
`"a %s parrot" % 'dead'`	Formatação de string

`s2.find('pa')`
`s2.replace('pa', 'xx')`
`s1.split()`
	Chamadas de método de string

`for x in s2`	Iteração,
`'m' in s2`	participação como membro

LITERAIS DE STRING

De modo geral, as strings são muito fáceis de usar no Python. Talvez o mais complicado a respeito delas seja que existem muitas maneiras de escrevê-las em seu código:

- Apóstrofos: `'spa"m'`
- Aspas: `"spa'm"`
- Apóstrofos ou aspas triplas: `'''... spam ...'''`, `"""... spam ..."""`
- Seqüências de escape: `"s\tp\na\om"`
- Strings brutas: `r"C:\new\test.spm"`
- Strings Unicode: `u'eggs\u0020spam'`

As formas de apóstrofos e aspas são, de longe, as mais comuns; as outras têm funções especializadas. Vamos ver rapidamente cada uma dessas opções.

Strings com apóstrofos e aspas são iguais

Em torno de strings do Python, caracteres com apóstrofos e aspas são intercambiáveis. Isto é, os literais de string podem ser escritos entre apóstrofos ou entre aspas – as duas formas funcionam de maneira igual e retornam o mesmo tipo de objeto. Por exemplo, uma vez escritas, as duas strings a seguir são idênticas:

```
>>> 'shrubbery', "shrubbery"
('shrubbery', 'shrubbery')
```

O motivo para incluir as duas é que isso permite que você incorpore um caractere de apóstrofo ou aspas do outro tipo dentro de uma string, sem fazer seu escape com uma barra invertida. Você pode incorporar um caractere de apóstrofo em uma string colocada entre aspas e vice-versa:

```
>>> 'knight"s', "knight's"
('knight"s', "knight's")
```

Incidentalmente, o Python concatena literais de string adjacentes automaticamente, embora seja quase tão simples adicionar um operador + entre elas, para ativar a concatenação explicitamente.

```
>>> title = "Meaning " 'of' " Life"
>>> title
'Meaning of Life'
```

Observe que, em todas essas saídas, o Python prefere imprimir strings com apóstrofos, a não ser que elas incorporem um. Você também pode incorporar aspas ou apóstrofos fazendo seu escape com barras invertidas:

```
>>> 'knight\'s', "knight\"s"
("knight's", 'knight"s')
```

Mas para entendermos o motivo, precisamos explicar como os escapes funcionam em geral.

Seqüências de escape definem bytes especiais

O último exemplo incorporou aspas ou apóstrofos dentro de uma string, precedendo-a com uma barra invertida. Isso é representativo de um padrão geral nas strings: barras invertidas são usadas para introduzir definições de byte especiais, conhecidas como *seqüências de escape*.

As seqüências de escape nos permitem incorporar códigos de byte em strings, que não podem ser digitados facilmente em um teclado. O caractere \, e um ou mais caracteres depois dele na literal de string, são substituídos por um único caractere no objeto string resultante, o qual tem o valor binário especificado pela seqüência de escape. Por exemplo, aqui está a string de cinco caracteres que incorpora uma nova linha e uma tabulação:

```
>>> s = 'a\nb\tc'
```

Os dois caracteres `\n` representam um único caractere – o byte que contém o valor binário do caractere de nova linha em seu conjunto de caracteres (normalmente, o código ASCII 10). Analogamente, a seqüência `\t` é substituída pelo caractere de tabulação. A maneira como a string aparece quando impressa depende de como você a imprime. O eco interativo mostra os caracteres especiais como escapes, mas a instrução `print`, em vez disso, os interpreta:

```
>>> s
'a\nb\tc'
>>> print s
a
b       c
```

Para ter certeza absoluta de quantos bytes existem nessa string, você pode usar a função interna `len` – ela retorna o número real de bytes em uma string, independente de como é exibida.

```
>>> len(s)
5
```

Essa string tem cinco bytes de comprimento: um byte "a" em ASCII, um byte de nova linha, um byte "b" em ASCII etc.; os caracteres de barra invertida originais não são armazenados com a string na memória.

Para codificar esses bytes especiais, o Python reconhece um conjunto completo de seqüências de código de escape, listadas na Tabela 5-2. Algumas seqüências permitem que você incorpore valores binários absolutos nos bytes de uma string. Por exemplo, aqui está outra string de cinco caracteres que incorpora dois bytes zero binários:

```
>>> s = 'a\0b\0c'
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

Tabela 5-2 Caracteres de barra invertida para string

Escape	Significado
`\newline`	Ignorado (continuação)
`\\`	Barra invertida (mantém uma \)
`\'`	Apóstrofo (mantém ')
`\"`	Aspas (mantém ")
`\a`	Bip
`\b`	Retrocesso
`\f`	Avanço de formulário
`\n`	Nova linha (Avanço de linha)
`\r`	Carriage return
`\t`	Tabulação horizontal
`\v`	Tabulação vertical
`\N{id}`	Id dbase Unicode
`\uhhhh`	Hexadecimal de 16 bits Unicode
`\Uhhhh...`	Hexadecimal de 32 bits Unicode[a]
`\xhh`	Valor de dígitos em hexadecimal hh
`\ooo`	Valor de dígitos em octal
`\0`	Nulo (não termina string)
`\other`	Não é escape (mantido)

[a] A seqüência de escape `\Uhhhh...` exige exatamente oito dígitos em hexadecimal (h); tanto `\u` como `\U` só podem ser usados em literais de string Unicode.

No Python, o byte zero (nulo) não termina uma string como normalmente acontece na linguagem C. Em vez disso, o Python mantém o comprimento e o texto da string na memória. Na verdade, nenhum caractere termina uma string no Python. Aqui está um exemplo onde tudo é código de escape binário absoluto – valores 1 e 2 em binário (escritos em octal), seguidos do valor 3 em binário (escrito em hexadecimal):

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
>>> len(s)
3
```

Isso se torna mais importante saber quando você processa arquivos de dados binários no Python. Como seu conteúdo é representado como string em seus scripts, está correto processar arquivos binários que contêm quaisquer tipos de valores de byte binários. Mais informações sobre arquivos aparecem no Capítulo 7.*

* Mas se você estiver particularmente interessado em arquivos de dados binários, a principal distinção é que eles são abertos no modo binário (use flags de modo de abertura com "b", como em "rb", "wb" etc.). Veja também o módulo `struct` padrão, que pode analisar dados binários carregados a partir de um arquivo.

Finalmente, conforme a última entrada da Tabela 5-2 sugere, se o Python não reconhece o caractere após um "\" como um código de escape válido, ele simplesmente mantém a barra invertida na string resultante:

```
>>> x = "C:\py\code"                    # mantém \ literalmente
>>> x
'C:\\py\\code'
>>> len(x)
10
```

A não ser que consiga memorizar tudo que há na Tabela 5-2, provavelmente você não deve contar com esse comportamento; para escrever barras invertidas literais, duplique ("\\" é um escape para "\") ou use strings brutas, descritas na próxima seção.

Strings brutas suprimem escapes

Conforme vimos, as seqüências de escape são úteis para incorporar códigos de bytes especiais dentro de strings. Às vezes, contudo, o tratamento especial das barras invertidas para introdução de escapes pode trazer problemas. É surpreendentemente comum, por exemplo, ver, em aula, iniciantes em Python tentando abrir um arquivo com um argumento de nome de arquivo semelhante ao seguinte:

```
myfile = open('C:\new\text.dat', 'w')
```

pensando que vão abrir um arquivo chamado *text.dat* em um diretório *C:\new*. O problema aqui é que `\n` é considerado um caractere de nova linha e `\t` é substituído por uma tabulação. Na verdade, a chamada tenta abrir um arquivo chamado *C:(novalinha)ew(tabulação)ext.dat*, normalmente com resultados errados.

É exatamente para esse tipo de coisa que as *strings brutas* são úteis. Se a letra "r" (maiúscula ou minúscula) aparece imediatamente antes das aspas ou apóstrofo de abertura de uma string, ela desativa o mecanismo de escape – o Python mantém suas barras invertidas literalmente, exatamente como você as digitou. Para corrigir o problema do nome de arquivo, basta lembrar de adicionar a letra "r", no Windows:

```
myfile = open(r'C:\new\text.dat', 'w')
```

Como duas barras invertidas são, na verdade, uma seqüência de escape para uma única barra invertida, você também pode manter suas barras invertidas simplesmente duplicando-as, sem usar strings brutas:

```
myfile = open('C:\\new\\text.dat', 'w')
```

Na verdade, o próprio Python às vezes usa esse esquema de duplicação ao imprimir strings com barras invertidas incorporadas:

```
>>> path = r'C:\new\text.dat'
>>> path                                # Mostra como código Python.
'C:\\new\\text.dat'
>>> print path                          # Formato amigável para o usuário
C:\new\text.dat
>>> len(path)                           # Comprimento da string
15
```

Na verdade, existe apenas uma barra invertida na string onde o Python imprimiu duas, na primeira saída desse código. Assim como na representação numérica, o formato padrão no prompt interativo imprime os resultados como se eles fossem código, mas a instrução `print`

fornece um formato mais amigável para o usuário. Para conferir, verifique o resultado da função interna `len` novamente, para ver o número de bytes na string, independente dos formatos de exibição. Se você contar, verá que na verdade há apenas um caractere por barra invertida, para um total de 15.

Além dos caminhos de diretório no Windows, as strings brutas também são normalmente usadas em expressões regulares (correspondência de padrão de texto, suportada com o módulo `re`); você conhecerá esse recurso posteriormente neste livro. Observe também que os scripts do Python normalmente podem usar barras *normais* em caminhos de diretório no Windows e no Unix, pois o Python tenta interpretar caminhos de forma portável. As strings brutas são úteis se você codifica caminhos usando barras invertidas nativas do Windows.

Aspas triplas definem strings de bloco de várias linhas

Até aqui, você viu apóstrofos, aspas, escapes e strings brutas. O Python também tem um formato de literal de string de aspas triplas, às vezes chamada de *string de bloco*, que é uma conveniência sintática para escrever dados textuais de várias linhas. Essa forma começa com três aspas (podem ser apóstrofos ou aspas mesmo), é seguida por qualquer número de linhas de texto e termina com a mesma seqüência de aspas triplas que a abriu. Apóstrofos e aspas no texto podem (mas não precisam) ter escapes. Por exemplo:

```
>>> mantra = """Always look
... on the bright
... side of life."""
>>>
>>> mantra
'Always look\n on the bright\nside of life.'
```

Essa string abrange três linhas (em algumas interfaces, o prompt interativo muda para "..." na continuação de linhas; o IDLE simplesmente desce uma linha). O Python reúne todo o texto entre aspas triplas em uma única string de várias linhas, com caracteres de nova linha (`\n`) incorporados nos lugares em que seu código tem quebras de linha. Note que a segunda linha no resultado tem um espaço inicial, como acontecia no literal – o que você digita é realmente o que obtém.

As strings de aspas triplas são úteis sempre que você precisa de texto de várias linhas em seu programa; por exemplo, para escrever mensagens de erro ou código em HTML ou XML. Você pode incorporar tais blocos diretamente em seu script, sem contar com arquivos de texto externos ou com caracteres de concatenação e de nova linha explícitos.

Strings em Unicode definem conjuntos de caracteres maiores

A última maneira de escrever strings em scripts talvez seja a mais especializada e a menos usada. As strings em Unicode normalmente são chamadas de strings de caractere "amplas". Como cada caractere pode ser representado por mais de um byte na memória, as strings em Unicode permitem que os programas codifiquem conjuntos de caracteres mais ricos do que as strings padrão.

Normalmente, as strings em Unicode são usadas para suportar a *internacionalização* de aplicativos (às vezes referida como "i18n", para compactar os 18 caracteres entre o primeiro e o último caractere do termo). Por exemplo, elas permitem que os programadores suportem diretamente conjuntos de caracteres europeus ou asiáticos em scripts Python. Como tais conjuntos de caracteres têm mais caracteres do que um único byte pode representar, o Unicode é normalmente usado para processar essas formas de texto.

No Python, as strings em Unicode podem ser escritas em seu script pela adição da letra "U" (minúscula ou maiúscula), imediatamente antes das aspas (ou apóstrofo) de abertura de uma string:

```
>>> u'spam'
u'spam'
```

Tecnicamente, essa sintaxe gera um objeto string em Unicode, que é um tipo de dados diferente das strings normais. Entretanto, o Python permite que você misture livremente strings em Unicode e normais nas expressões e converte para Unicode os resultados de tipo misto.

```
>>> 'ni' + u'spam'                # Tipos de string mistos
u'nispam'
```

Na verdade, as strings em Unicode são definidas para suportar todas as operações normais de processamento de string que você encontrará na próxima seção, de modo que a diferença nos tipos é freqüentemente trivial para seu código. Assim como as strings normais, as strings em Unicode podem ser concatenadas, indexadas, fracionadas, combinadas com o módulo `re` etc., e não podem ser alteradas no local. Se você precisar mesmo fazer a conversão entre os dois tipos explicitamente, pode usar as funções internas `str` e `unicode`:

```
>>> str(u'spam')                  # De Unicode para normal
'spam'
>>> unicode('spam')               # De normal para unicode
u'spam'
```

Como o Unicode é projetado para manipular caracteres de vários bytes, você também pode usar os escapes especiais `\u` e `\U` para codificar valores de caractere binários maiores do que 8 bits:

```
>>> u'ab\x20cd'                   # caracteres de 8 bits\1 byte
u'ab cd'
>>> u'ab\u0020cd'                 # caracteres de 2 bytes
u'ab cd'
>>> u'ab\U00000020cd'             # caracteres de 4 bytes
u'ab cd'
```

O primeiro deles incorpora o código binário de um caractere de espaço; seu valor binário em notação hexadecimal é `x20`. O segundo e o terceiro fazem o mesmo, mas fornecem o valor em notação de escape Unicode de 2 e 4 bytes.

Mesmo que você não ache que vai precisar de caracteres Unicode, poderá usá-los sem saber. Como algumas interfaces de programação (por exemplo, a API COM no Windows) representam texto como Unicode, esses caracteres podem aparecer em seu script como entradas ou resultados de API e, às vezes, você precisa converter entre os tipos normal e Unicode. Como o Python trata dos dois tipos de string indistintamente na maioria dos contextos, a presença de strings em Unicode é freqüentemente transparente para seu código – de modo geral, você pode ignorar o fato de que o texto está sendo passado como objetos Unicode e usar operações de strings normais.

O conjunto de caracteres Unicode é um acréscimo útil no Python; como ele é incorporado, é fácil manipular tais dados em seus scripts, quando necessário. Infelizmente, deste ponto em diante, a história do Unicode se torna muito complexa. Por exemplo:

- Os objetos Unicode fornecem um método `encode` que converte uma string em Unicode para uma string normal de 8 bits, usando uma codificação específica.

- A função interna `unicode` e o módulo `codecs` suportam "codecs" (de COders e DECoders – codificadores e decodificadores) Unicode registrados.
- O módulo `unicodedata` dá acesso ao banco de dados de caracteres Unicode.
- O módulo `sys` inclui chamadas para buscar e configurar o esquema de codificação padrão Unicode (normalmente, o padrão é ASCII).
- Você pode combinar os formatos de string bruta e unicode (por exemplo, `ur'a\b\c'`).

Como o conjunto de caracteres Unicode é uma ferramenta relativamente avançada e raramente usada, omitiremos outros detalhes neste texto introdutório. Consulte o manual padrão do Python para conhecer o resto da história do Unicode.

STRINGS EM AÇÃO

Quando você tiver escrito uma string, provavelmente desejará usá-la. Esta seção e as próximas duas demonstram os fundamentos, formatação e métodos de string.

Operações básicas

Vamos começar pela interação com o interpretador do Python, para ilustrarmos as operações de string básicas listadas na Tabela 5-1. As strings podem ser concatenadas usando-se o operador `+`, e repetidas usando-se o operador `*`:

```
% pyton
>>> len('abc')           # Comprimento: itens numéricos
3
>>> 'abc' + 'def'        # Concatenação: uma nova string
'abcdef'
>>> 'Ni!' * 4            # Repetição: como "Ni!" + "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

Formalmente, somar dois objetos string cria um novo objeto string com o conteúdo de seus operandos unidos; a repetição é como somar várias vezes uma string com ela mesma. Nos dois casos, o Python permite que você crie strings de tamanho arbitrário; não há necessidade de declarar nada previamente no Python, incluindo os tamanhos das estruturas de dados.* A função interna `len` retorna o comprimento de strings (e de outros objetos que tenham um comprimento).

A repetição pode parecer um pouco obscura à primeira vista, mas se mostra útil em um número de contextos surpreendente. Por exemplo, para imprimir uma linha de 80 traços, você pode contar até 80 ou deixar o Python contar em seu lugar:

```
>>> print '------- ...seqüência... ---'       # 80 traços do jeito difícil
>>> print '-'*80                              # 80 traços do jeito fácil
```

* Ao contrário dos arrays de caracteres da linguagem C, você não precisa alocar nem gerenciar arrays de armazenamento ao usar strings no Python. Basta criar objetos string conforme for necessário e deixar o Python gerenciar o espaço de memória subjacente. O Python recupera automaticamente o espaço de memória não utilizado dos objetos, usando uma estratégia de coleta de lixo com contagem de referência. Cada objeto monitora o número de nomes, estruturas de dados etc. que referencia; quando a contagem chega a zero, o Python libera o espaço do objeto. Esse esquema significa que o Python não precisa parar e varrer toda a memória para encontrar espaço não utilizado para liberar (um componente de lixo adicional também coleta objetos cíclicos).

Note que a sobrecarga de operador já está em funcionamento aqui: estamos usando os mesmos operadores + e *, que são chamados de adição e multiplicação quando usamos números. O Python efetua a operação correta, pois conhece os tipos dos objetos que estão sendo somados e multiplicados. Mas, tome cuidado: isso não é tão liberal quanto você poderia esperar. Por exemplo, o Python não permite que você misture números e strings em expressões com o operador +: `'abc' +9` gera um erro, em vez de converter `9` para uma string automaticamente.

Conforme mostrado na última linha da Tabela 5-1, você também pode fazer uma iteração sobre strings em loops usando instruções `for` e testar a participação como membro com o operador de expressão `in`, o que é basicamente uma pesquisa:

```
>>> myjob = "hacker"
>>> for c in myjob: print c,          # Percorre os itens.
...
h a c k e r
>>> "k" in myjob                      # 1 significa true (encontrado).
1
>>> "z" in myjob                      # 0 significa false (não encontrado)
0
```

O loop `for` atribui uma variável a sucessivos itens em uma seqüência (aqui, uma string) e executa uma ou mais instruções para cada item. Na verdade, aqui, a variável `c` se torna um cursor percorrendo a string. Outros detalhes sobre esses exemplos serão discutidos posteriormente.

Indexação e fracionamento

Como as strings são definidas como uma coleção ordenada de caracteres, podemos acessar seus componentes pela posição. No Python, os caracteres de uma string são buscados por *indexação* – fornecendo o deslocamento numérico do componente desejado, entre colchetes, após a string. Você recebe a string de um único caractere.

Assim como na linguagem C, os deslocamentos do Python começam em zero e terminam em um a menos do que o comprimento da string. Ao contrário da linguagem C, o Python também permite que você busque itens a partir de seqüências, como em strings usando deslocamentos *negativos*. Tecnicamente, deslocamentos negativos são somados ao comprimento de uma string para deduzir um deslocamento positivo. Você também pode considerar os deslocamentos negativos como uma contagem regressiva a partir do final.

```
>>> S = 'spam'
>>> S[0], S[-2]                # Indexação a partir do início ou do fim
('s', 'a')
>>> S[1:3], S[1:], S[:-1]      # Fracionamento: extrai seção
('pa', 'pam', 'spa')
```

A primeira linha define uma string de quatro caracteres e atribui a ela o nome `S`. A linha seguinte a indexa de duas maneiras: `S[0]` busca o item no deslocamento `0` a partir da esquerda (a string de um caractere 's') e `S[-2]` obtém o item no deslocamento `2` a partir do fim (ou equivalentemente, no deslocamento (`4 + -2`) a partir do início). Os deslocamentos e fracionamentos são mapeados em células, como mostrado na Figura 5-1.*

* Os leitores mais ligados à matemática (e os alunos em minhas aulas) às vezes detectam uma pequena assimetria aqui: o item mais à esquerda está no deslocamento 0, mas o mais à direita está no deslocamento –1. Pois é, não existe nenhum valor –0 distinto no Python.

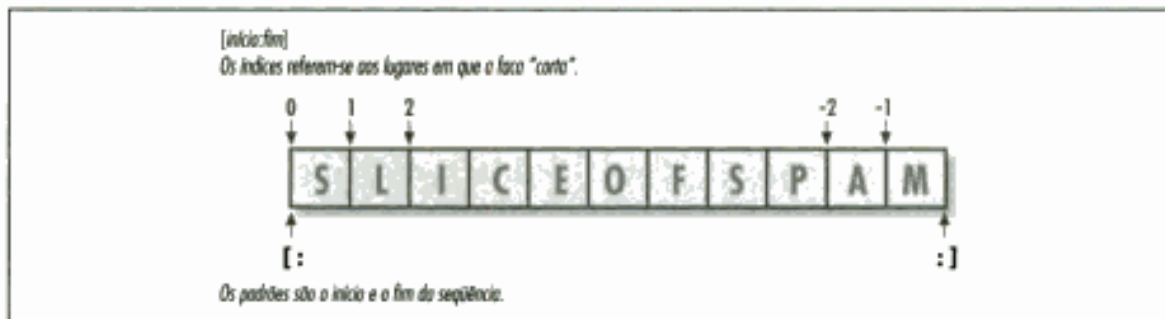



Figura 5-1 Usando deslocamentos e fracionamentos.

A última linha no exemplo anterior representa a primeira vez que vemos o *fracionamento*. Provavelmente, a melhor maneira de pensar no fracionamento seja como uma forma de *análise* (estrutura de análise), especialmente quando aplicado às strings – ele nos permite extrair uma seção (substring) inteira de uma só vez. Os fracionamentos podem extrair colunas de dados, cortar texto inicial e final, e muito mais.

Aqui está como o fracionamento funciona. Quando você indexa um objeto seqüência, como uma string, em um par de deslocamentos separados por dois-pontos, o Python retorna um novo objeto contendo a seção adjacente identificada pelo par de deslocamentos. O deslocamento à esquerda é considerado o limite inferior (incluído) e o da direita é o limite superior (não incluído). O Python busca todos os itens, do limite inferior até o limite superior (não incluído), e retorna um novo objeto contendo os itens buscados. Se forem omitidos, os limites da esquerda e da direita terão como padrão, respectivamente, zero e o comprimento do objeto que você está fracionando.

Assim, no exemplo anterior, `S[1:3]` extrai itens como os deslocamentos 1 e 2. Ele pega o segundo e o terceiro itens e pára antes do quarto, no deslocamento 3. Em seguida, `S[1:]` obtém *todos os itens após o primeiro* – o limite superior tem como padrão o comprimento da string. Finalmente, `S[:-1]` busca *todos os itens, menos o último* – o limite inferior tem zero como padrão e –1 refere-se ao último item, não incluído.

Isso pode parecer confuso à primeira vista, mas a indexação e o fracionamento são simples e poderosos para se usar, uma vez que você pegue o jeito. Lembre-se de que, se você não tiver certeza a respeito do que um fracionamento significa, experimente fazer isso interativamente. No próximo capítulo, você verá que também é possível alterar uma seção inteira de determinado objeto de uma só vez, atribuindo a um fracionamento. Aqui está um resumo dos detalhes, para referência:

A indexação (S[i]) busca componentes em deslocamentos

- O primeiro item está no deslocamento 0.
- Índices negativos significam contagem regressiva a partir do fim ou da direita.
- `S[0]` busca o primeiro item.
- `S[-2]` busca o segundo item a partir do fim (como `S[len(S)-2]`).

O fracionamento (S[i:j]) extrai seções adjacentes de uma seqüência

- O limite superior é não-inclusivo.
- Os limites do fracionamento terão 0 e o comprimento da seqüência como padrão, se forem omitidos.

- `S[1:3]` busca do deslocamento 1 até, mas não incluindo, 3.
- `S[1:]` busca do deslocamento 1 até o fim (comprimento).
- `S[:3]` busca do deslocamento 0 até, mas não incluindo, 3.
- `S[:-1]` busca do deslocamento 0 até, mas não incluindo, o último item.
- `S[:]` busca do deslocamento 0 até o fim – uma cópia de nível superior de `S`.

Veremos outro exemplo de fracionamento como ferramenta de análise, posteriormente nesta seção. O último item listado aqui mostra um truque muito comum: ele faz uma *cópia* de nível superior completa de um objeto de seqüência – um objeto com o mesmo valor, mas uma parte diferente da memória. Isso não é muito útil para objetos imutáveis, como as strings, mas é interessante para objetos que podem ser alterados, como as listas (mais informações sobre cópias aparecem no Capítulo 7). Posteriormente, veremos ainda que a sintaxe usada para indexar pelo deslocamento (os colchetes) também é usada para indexar dicionários pela chave; as operações parecem iguais, mas têm interpretações diferentes.

No Python 2.3, as expressões de fracionamento suportam um terceiro índice opcional, usado como passo (às vezes chamado de pernada). O passo é adicionado no índice de cada item extraído. Por exemplo, `X[1:10:2]` buscará cada *outro* item em `X`, a partir dos deslocamentos 1-9; isso coletará itens a partir dos deslocamentos 1, 3, 5 etc. Analogamente, a expressão de fracionamento `"hello"[::-1]` retorna a nova string `"olleh"`. Para saber mais detalhes, consulte a documentação padrão do Python ou faça algumas experiências interativamente.

Por que isto é relevante: fracionamentos

Em todas as partes deste livro relacionadas à linguagem básica, incluímos quadros como este para que você veja como alguns dos recursos que estão sendo apresentados são normalmente usados em programas reais. Como não podemos mostrar muito uso real, até você ter visto a maior parte do quadro geral do Python, esses quadros necessariamente contêm muitas referências a assuntos ainda não apresentados; você deve considerá-los, no máximo, como uma prévia das maneiras pelas quais poderá achar úteis esses conceitos abstratos da linguagem, para tarefas de programação comuns.

Por exemplo, você verá posteriormente que as palavras de argumento listadas em uma linha de comando, usadas para executar um programa em Python, se tornam disponíveis no atributo argv do módulo interno `sys`:

```
# Arquivo echo.py
import sys
print sys.argv
% python echo.py -a -b -c
['echo.py', '-a', '-b', '-c']
```

Normalmente, você só está interessado em inspecionar os argumentos passados após o nome do programa. Isso leva a uma aplicação muito típica dos fracionamentos: uma única expressão de fracionamento pode cortar todos os itens da lista, menos o primeiro. Aqui, `sys.argv[1:]` retorna a lista desejada, `['-a', '-b', '-c']`. Você pode, então, processar sem ter de acomodar o nome do programa na frente.

Os fracionamentos também são freqüentemente usados para limpar linhas de arquivos de entrada; se você sabe que uma linha terá um caractere de final de linha no fim (um marcador de nova linha `'\n'`), pode desfazer-se dele com uma única expressão como `line[:-1]`, que extrai todos os caracteres, menos o último da linha (o limite inferior tem 0 como padrão). Nos dois casos, os fracionamentos fazem o trabalho da lógica, que deve ser explícita em uma linguagem de nível mais baixo.

Ferramentas de conversão de string

Você não pode somar um número e uma string no Python, mesmo que a string pareça um número (isto é, seja toda composta de dígitos):

```
>>> "42" + 1
TipeError: cannot concatenate 'str' and 'int' objects
```

Isso é assim por design: como + pode significar adição e concatenação, a escolha da conversão seria ambígua. Portanto, o Python trata isso como um erro. No Python, mágica geralmente é omitida, caso ela torne sua vida mais complexa.

O que fazer, então, se seu script recebe um número como uma string de texto de um arquivo ou da interface com o usuário? O truque é que você precisa empregar ferramentas de conversão antes de tratar uma string como um número ou vice-versa. Por exemplo:

```
>>> int("42"), str(42)              # Converte de/para string
(42, '42')
>>> string.atoi("42"), '42'         # O mesmo, mas são técnicas mais
                                      antigas
(42, '42')
```

As funções `int` e `string.atoi` convertem uma string em um número; a função `str` e *acentos graves* em torno de qualquer objeto convertem esse objeto em sua representação de string (por exemplo, `` `42` `` converte um número em uma string). Dessas, `int` e `str` são as técnicas de conversão mais recentes, geralmente prescritas, e não exigem importação do módulo string.

Embora você não possa misturar strings e tipos numéricos em torno de operadores como +, se necessário, pode fazer a conversão manualmente antes dessa operação:

```
>>> int("42") + 1                   # Força a adição
43
>>> "spam" + str(42)                # Força a concatenação
'spam42'
```

Funções internas semelhantes manipulam conversões de número em ponto flutuante:

```
>>> str(3.1415), float("1.5")
('3.1415', 1.5)

>>> text = "1.234E-10"
>>> float(text)
1.2340000000000001e-010
```

Posteriormente, estudaremos melhor a função interna `eval`; ela executa uma string contendo código de expressão Python e, portanto, pode converter uma string para qualquer tipo de objeto. As funções `int`, `string.atoi` e suas parentes convertem apenas para números, mas essa restrição significa que, normalmente, elas são mais rápidas. Conforme foi visto no Capítulo 4, a expressão de formatação de string fornece outra maneira de converter números em strings.

Alterando strings

Lembra-se do termo "seqüência imutável"? A parte imutável significa que você não pode alterar uma string no local (por exemplo, atribuindo a um índice):

```
>>> S = 'spam'
>>> S[0] = "x"
Acarreta um erro!
```

Então, como você modifica informações em texto no Python? Para alterar uma string, você precisa apenas construir e atribuir uma nova string, usando ferramentas como concatenação e fracionamento, e possivelmente atribuindo o resultado de volta ao nome original da string.

```
>>> S = s + 'SPAM!'          # Para alterar uma string, faça uma nova.
>>> S
'spamSPAM!'
>>> S = S[:4] + 'Burger' + S[-1]
>>> S
'spamBurger!'
```

O primeiro exemplo adiciona uma substring no final de S, por meio de concatenação; na verdade, ele faz uma nova string e a atribui de volta a S, mas você normalmente pode considerar isso como a alteração de uma string. O segundo exemplo substitui quatro caracteres por seis, por meio de fracionamento, indexação e concatenação. Posteriormente nesta seção, você verá como fazer para obter um efeito semelhante com chamadas de método de string. Finalmente, também é possível construir novos valores de texto com expressões de formatação de string:

```
>>> 'That is %d %s bird!' % (1, 'dead')        # como a instrução sprintf da
                                                  linguagem C
That is 1 dead bird!
```

A próxima seção mostra como.

FORMATAÇÃO DE STRING

O Python utiliza o operador binário % para trabalhar em strings (o operador % também significa módulo de resto de divisão, para números). Quando aplicado a strings, ele tem o mesmo emprego da função sprintf da linguagem C; o operador % oferece uma maneira simples de formatar valores como strings, de acordo com uma string de definição de formato. Em resumo, esse operador fornece uma maneira compacta de escrever várias substituições de string.

Para formatar strings:

1. Forneça uma string de formato à esquerda do operador %, com destinos de conversão incorporados que comecem com % (por exemplo, "%d").
2. Forneça um objeto (ou objetos entre parênteses) à direita do operador %, que você queira que o Python insira na string de formato à esquerda, em seus destinos de conversão.

Assim, no último exemplo da seção anterior, o inteiro 1 substitui %d na string de formato à esquerda e a string 'dead' substitui %s. O resultado é uma nova string que reflete essas duas substituições.

Tecnicamente falando, a expressão de formatação de string normalmente é opcional – geralmente você pode fazer um trabalho semelhante com várias concatenações e conversões. Entretanto, a formatação nos permite combinar muitos passos em uma única operação. Isso é poderoso o suficiente para garantir mais alguns exemplos:

```
>>> exclamation = "Ni"
>>> "The knights who say %s!" % exclamation
'The knights who say Ni!'
>>> "%d %s %d you" % (1, 'spam', 4)
'1 spam 4 you'
>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])
'42 -- 3.14159 -- [1, 2, 3]'
```

Aqui, o primeiro exemplo liga a string `"Ni"` ao destino à esquerda, substituindo o marcador `%`. No segundo, três valores são inseridos na string de destino. Quando há mais de um valor sendo inserido, você precisa agrupar os valores à direita, entre parênteses (o que significa, na realidade, que eles são colocados em uma tupla). Lembre-se de que a formatação sempre produz uma nova string, em vez de alterar a que está à esquerda; como as strings são imutáveis, isso é necessário.

Note que o terceiro exemplo insere três valores novamente – um inteiro, outro de ponto flutuante e um objeto lista –, mas todos os seus destinos à esquerda são `%s`, que significa conversão para string. Como todo tipo de objeto pode ser convertido para uma string (que é usada para impressão), todo objeto funciona com o código de conversão `%`. Por isso, a não ser que você vá fazer alguma formatação especial, `%s` é freqüentemente o único código que precisa lembrar.

Formatação de string avançada

Para uma formatação mais avançada específica de tipo, você pode usar qualquer um dos códigos de conversão listados na Tabela 5-3 em expressões de formatação. Os programadores de C reconhecerão a maioria deles, pois a formatação de string do Python suporta todos os códigos de formato de `printf` normais da linguagem C (mas retornam o resultado, em vez de exibi-lo, como faz `printf`). Alguns dos códigos de formato da tabela fornecem maneiras alternativas de formatar o mesmo tipo; por exemplo, `%e`, `%f` e `%g` fornecem maneiras alternativas de formatar números de ponto flutuante.

Tabela 5-3 Códigos de formatação de string

Código	Significado
`%s`	String (ou qualquer objeto)
`%r`	`s`, mas usa `repr()` e não `str()`
`%c`	Caractere
`%d`	Decimal (inteiro)
`%i`	Inteiro
`%u`	Sem sinal (inteiro)
`%o`	Inteiro em octal
`%x`	Inteiro em hexadecimal
`%X`	x, mas imprime em maiúscula
`%e`	Expoente de ponto flutuante
`%E`	e, mas imprime em maiúscula
`%f`	Decimal de ponto flutuante
`%g`	e ou f de ponto flutuante
`%G`	E ou f de ponto flutuante
`%%`	`'%'` literal

Na verdade, os destinos da conversão de string no formato do lado esquerdo da expressão suportam uma variedade de operações de conversão, com uma sintaxe própria bastante sofisticada. A estrutura geral dos destinos de conversão é a seguinte:

```
%[(nome)][flags][largura][.precisão]código
```

Os códigos de caractere da Tabela 5-3 aparecem no final da string de destino. Entre `%` e o código de caractere, podemos fornecer uma chave de dicionário, listar flags que especificam

coisas como justificação à esquerda (-), sinal numérico (+) e preenchimentos com zero (0), fornecer a largura total do campo e o número de dígitos após um ponto decimal e muito mais.

A sintaxe do destino da formatação está completamente documentada em outra parte, mas para demonstrar a sintaxe de formato usada comumente, aqui estão alguns exemplos. O primeiro formata inteiros por padrão e, então, em um campo de seis caracteres com justificação à esquerda e preenchimento com zero:

```
>>> x = 1234
>>> res = "integers:...%d...%-6d...%06d" % (x, x, x)
>>> res
'integers: ...1234...1234 ...001234'
```

Os formatos %e,%f e %g exibem números de ponto flutuante de diferentes maneiras, como:

```
>>> x = 1.23456789
>>> x
1.2345678899999999

>>> '%e | %f | %g' % (x, x, x)
'1.234568e+000 | 1.234568 | 1.23457'
```

Para números de ponto flutuante, podemos obter uma variedade de efeitos de formatação adicionais, especificando justificação à esquerda, preenchimento com zero, sinais numéricos, largura do campo e dígitos após o ponto decimal. Para tarefas mais simples, você pode simplesmente converter para strings, com uma expressão de formato ou com a função interna str, mostrada anteriormente:

```
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23 | 01.23 | +001.2'

>>> "%s" % x, str(x)
('1.23456789', '1.23456789')
```

A formatação de strings também permite que os destinos de conversão à esquerda se refiram a chaves em um *dicionário* à direita, para buscar o valor correspondente. Não falamos muito sobre dicionários ainda, mas aqui estão os fundamentos para referência futura:

```
>>> "%(n)d %(x)s" % {"n":1, "x":"spam"}
'1 spam'
```

Aqui, (n) e (x) na string de formato referem-se a chaves na literal de dicionário à direita e buscam seus valores associados. Esse truque é freqüentemente usado em conjunto com a função interna vars, que retorna um dicionário contendo todas as variáveis que existem no local em que ela é chamada:

```
>>> food = 'spam'
>>> age = 40
>>> vars()
{'food': 'spam', 'age': 40,...muito mais... }
```

Quando usado à direita de uma operação de formato, isso permite que a string de formato se refira às variáveis pelo nome (isto é, pela chave de dicionário):

```
>>> "(age)d %(food)s" % vars()
'40 spam'
```

Falaremos muito mais sobre dicionários no Capítulo 6. Consulte também a seção "Números", no Capítulo 4, para ver exemplos que convertem para strings em hexadecimal e em octal com os códigos de destino de formatação %x e %o.

MÉTODOS DE STRING

Além dos operadores de expressão, as strings fornecem um conjunto de *métodos* que implementam tarefas de processamento de texto mais sofisticadas. Os métodos são simplesmente funções associadas a um objeto em particular. Tecnicamente, eles são atributos anexados aos objetos, que referenciam uma função que pode ser chamada. No Python, os métodos são específicos dos tipos de objeto; os métodos de string, por exemplo, só funcionam em objetos string.

As funções são pacotes de código e as chamadas de método combinam duas operações ao mesmo tempo – uma busca de atributo e uma chamada:

Buscas de atributo
: Uma expressão na forma `objeto.atributo` significa "buscar o valor de atributo em objeto".

Expressões de chamada
: Uma expressão na forma `função(argumentos)` significa "ative o código de função, passando a ela zero ou mais objetos argumento separados por vírgulas e retornando o valor do resultado da função".

Colocar esses dois itens juntos nos permite chamar um método de um objeto. A expressão de chamada de método `objeto.método(argumentos)` é avaliada da esquerda para a direita – o Python buscará primeiro o método do objeto e depois o chamará, passando os argumentos. Se o método calcula um resultado, ele voltará como o resultado da expressão de chamada de método inteira.

Conforme você verá em toda a Parte II, a maioria dos objetos tem métodos que podem ser chamados e todos são acessados usando-se essa mesma sintaxe de chamada de método. Para chamar um método de objeto, você tem de passar por um objeto existente; vamos ver alguns exemplos para saber como se faz isso.

Exemplos de método de string: alterando strings

A Tabela 5-4 resume os padrões de chamada de métodos de string internos. Eles implementam operações de nível mais alto, como divisão e junção, conversões de caixa e testes, e pesquisas de substring. Vamos trabalhar em código que demonstra alguns dos métodos mais comumente usados em ação e, no processo, apresentar os fundamentos do processamento de texto do Python.

Tabela 5-4 Chamadas de método de string

`S.capitalize()`	`S.ljust(width)`
`S.center(width)`	`S.lower()`
`S.count(sub [, start [ end]])`	`S.lstrip()`
`S.encode([encoding [,errors]])`	`S.replace(old, new [, maxsplit])`
`S.endswith(suffix [, start [, end]])`	`S.rfind(sub [,start [,end]])`
`S.expandtabs([tabsize])`	`S.rindex(sub [, start [, end]])`
`S.find(sub [, start [, end]])`	`S.rjust(width)`
`S.index(sub [, start [, end]])`	`S.rstrip()`
`S.isalnum()`	`S.split([sep [,maxsplit]])`
`S.isalpha()`	`S.splitlines([keepends])`
`S.isdigit()`	`S.startwith(prefix [, start [, end]])`
`S.islower()`	`S.strip()`
`S.isspace()`	`S.swapcase()`
`S.istitle()`	`S.title()`
`S.isupper()`	`S.translate(table [, delchars])`
`S.join(seq)`	`S.upper()`

Como as strings são imutáveis, elas não podem ser alteradas no local diretamente. Para compor um novo valor de texto, você sempre pode construir uma nova string com operações como fracionamento e concatenação. Por exemplo, para substituir dois caracteres no meio de uma string:

```
>>> S = 'spammy'
>>> S = S[:3] + 'xx' + S[5:]
>>> S
'spaxxy'
```

Mas, se você quiser realmente substituir uma substring, pode usar o método de string `replace`:

```
>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')
>>> S
'spaxxy'
```

O método `replace` é mais geral do que esse código sugere. Ele recebe como argumentos a substring original (de qualquer comprimento) e a string (de qualquer comprimento) que vai substituí-la, realizando uma busca e troca global:

```
>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')
'aaSPAMbbSPAMccSPAMdd'
```

Em tais funções, `replace` pode ser usado para implementar tipos de ferramenta de substituição de modelo (por exemplo, cartas padronizadas). Observe como desta vez o resultado simplesmente imprime, em vez de atribuir um nome; você só precisa atribuir os resultados a nomes se quiser mantê-los para uso posterior. Se você precisar substituir uma única string de tamanho fixo, que pode ocorrer em qualquer deslocamento, pode fazer a substituição novamente ou procurar a substring com o método de string `find` e fracionar:

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> where = S.find('SPAM')            # Procura posição
>>> where                             # Ocorre no deslocamento 4
4
>>> S = S[:where] + 'EGGS' + S[(where+4):]
>>> S
'xxxxEGGSxxxxSPAMxxxx'
```

O método `find` retorna o deslocamento onde a string aparece (por padrão, pesquisa a partir da frente) ou –1, caso ela não seja encontrada. Outra maneira é usar replace com um terceiro argumento para limitá-la a uma única substituição:

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> S.replace('SPAM', 'EGGS')            # Substitui tudo
'xxxxEGGSxxxxEGGSxxxx'

>>> S.replace('SPAM','EGGS', 1)          # Substitui uma
'xxxxEGGSxxxxSPAMxxxx'
```

Note que, aqui, `replace` está retornando uma nova string a cada vez. Como as strings são imutáveis, os métodos nunca alteram a string submetida no local, mesmo que eles se chamem "replace" (substituir).

Na verdade, um inconveniente em potencial do uso de concatenação ou do método `replace` para alterar strings é que ambos geram novos objetos string sempre que são executados. Se você tiver que aplicar muitas alterações em uma string muito grande, talvez tenha que me-

lhorar o desempenho do seu script, convertendo a string em um objeto que suporte alterações no local:

```
>>>> S = 'spammy'
>>> L = list(S)
>>> L
['s', 'p', 'a', 'm', 'm', 'y']
```

A função interna `list` (ou uma chamada de construção de objeto) constrói uma nova lista a partir dos itens de qualquer seqüência – neste caso, "explodindo" os caracteres de uma string em uma lista.

Uma vez nessa forma, você pode fazer várias alterações, sem gerar cópias da string para cada alteração:

```
>>> L[3] = 'x'                # Funciona para listas e não para strings
>>> L[4] = 'x'
>>> L
['s', 'p', 'a', 'x', 'x', 'y']
```

Se, após suas alterações, você precisar converter de volta para uma string (por exemplo, para gravar em um arquivo), use o método de string `join` para "implodir" a lista de volta em uma string:

```
>>> S = ' '.join(L)
>>> S
'spaxxy'
```

O método `join` pode parecer um pouco invertido à primeira vista. Como se trata de um método de strings (e não a lista de strings), ele é chamado por meio do delimitador desejado. `join` reúne as strings da lista, com o delimitador entre os itens da lista; neste caso, usando um delimitador de string vazio para converter a lista de volta para string. Em geral, qualquer delimitador de string e lista de strings servirá:

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast'
```

Exemplos de método de string: análise de texto

Outra função comum dos métodos de string é como uma forma simples de *análise* de texto – analisar a estrutura e extrair substrings. Para extrair substrings em deslocamentos fixos, podemos empregar técnicas de fracionamento:

```
>>> line = 'aaa bbb ccc'
>>> col1 = line[0:3]
>>> col3 = line[8:]
>>> col1
'aaa'
>>> col3
'ccc'
```

Aqui, as colunas de dados aparecem em deslocamentos fixos e, assim, podem ser fracionadas a partir da string original. Essa técnica passa por análise, contanto que seus dados tenham posições fixas para seus componentes. Se, em vez disso, os dados são separados por algum tipo de delimitador, podemos extrair seus componentes por meio de fracionamento, mesmo que os dados possam aparecer em posições arbitrárias dentro da string:

```
>>> line = 'aaa bbb ccc'
>>> cols = line.split()
>>>cols
['aaa', 'bbb', 'ccc']
```

O método de string `split` corta uma string transformando-a em uma lista de substrings, em torno de um delimitador. Não passamos um delimitador no exemplo anterior, de modo que ele assume o padrão de um espaço em branco – a string é dividida em grupos de um ou mais espaços, tabulações ou caracteres de nova linha, e obtemos de volta uma lista das substrings resultantes. Em outras aplicações, os dados podem ser separados por delimitadores mais tangíveis, como palavras-chave ou vírgulas:

```
>>> line = 'bob,hacker,40'
>>> line.split(',')
['bob', 'hacker', '40']
```

Esse exemplo divide (e, portanto, analisa) a string em vírgulas, um separador comum em dados retornados por algumas ferramentas de banco de dados. Os delimitadores também podem ser mais longos do que um único caractere:

```
>>> line = "i'mSPAMaSPAMlumberjack"
>>> line.split("SPAM")
["i'm", 'a', 'lumberjack']
```

Embora haja limites para a análise potencial de fracionamento e divisão, ambas são executados muito rapidamente e podem tratar de tarefas básicas de extração de texto.

Você vai encontrar mais exemplos de string posteriormente neste livro. Para obter mais detalhes, consulte também o manual da biblioteca Python e outras fontes de documentação, ou simplesmente experimente isso interativamente, por conta própria. Note que nenhum dos métodos de string aceita *padrões* – para processamento de texto baseado em padrão, você deve usar o módulo de biblioteca padrão `re` do Python. Contudo, por causa dessa limitação, às vezes os métodos de string são executados mais rapidamente do que as ferramentas do módulo `re`.

O módulo original

A história do método de string do Python foi um tanto distorcida pelo tempo. Aproximadamente na primeira década da existência do Python, ele fornecia um módulo de biblioteca padrão chamado `string`, o qual continha funções que, de modo geral, espelhavam o conjunto atual de métodos de objeto string. Posteriormente, no Python 2.0 (e no 1.6, que teve vida curta), essas funções se tornaram disponíveis como métodos de objetos string, em resposta aos pedidos dos usuários. Como muitas pessoas escreviam códigos que contavam com o módulo string original, ele foi mantido para compatibilidade com versões anteriores.

O resultado desse legado é que, hoje, existem normalmente duas maneiras de executar operações de string avançadas – chamando-se métodos de objeto ou chamando-se funções de módulo string e passando o objeto como argumento. Por exemplo, dada uma variável `X` atribuída a um objeto string, chamar um método de objeto:

```
X.método(argumentos)
```

normalmente é equivalente a chamar a mesma operação por meio do módulo:

```
string.método(X, argumentos)
```

desde que você já tenha importado o módulo string. Aqui está um exemplo dos dois padrões de chamada em ação – primeiro, o esquema do método:

```
>>> S = 'a+b+c+'
>>> x = S.replace('+', 'spam')
>>> x
'aspambspamcspam'
```

Para acessar a mesma operação por meio do módulo, você precisa importar o módulo (pelo menos uma vez em seu processo) e passar o objeto:

```
>>> import string
>>> y = string.replace(S, '+', 'spam')
>>> y
'aspambspamcspam'
```

Como a estratégia do módulo foi o padrão por muito tempo e como as strings são componentes básicos da maioria dos programas, você provavelmente verá os dois padrões de chamada em código Python que encontrar.

Atualmente, contudo, a recomendação geral é usar métodos, em vez do módulo. O esquema de chamada de módulo exige que você importe o módulo `string` (os métodos não exigem). O módulo `string` exige a digitação de alguns caracteres a mais nas chamadas (pelo menos quando você carrega o módulo com `import`, mas não para `from`). Além disso, o módulo pode ser executado mais lentamente dos que os métodos (o módulo corrente faz o mapeamento da maioria das chamadas para os métodos e, assim, acarreta uma chamada extra no caminho).

Por outro lado, como a sobreposição entre ferramentas de módulo e método não é exata, às vezes talvez ainda seja necessário usar um ou outro esquema – alguns métodos só estão disponíveis como métodos mesmo e alguns, como funções de módulo. Além disso, alguns programadores preferem usar o padrão de chamada de módulo, pois o nome do módulo torna mais evidente que o código está chamando ferramentas de string: para alguns, `string.método(x)` parece mais auto-documentado do que `x.método()`. Como sempre, em última análise, a escolha é sua.

CATEGORIAS DE TIPO GERAIS

Agora que já vimos o primeiro objeto de coleção, a string, vamos fazer uma pausa para definir alguns conceitos de tipo geral que se aplicam à maioria dos tipos daqui em diante. Com referência aos tipos internos, verifica-se que as operações funcionam igualmente para todos os tipos de uma categoria; portanto, precisamos definir a maioria das idéias apenas uma vez. Até aqui, vimos apenas números e strings, mas como eles representam duas das três categorias de tipo principais no Python, você já sabe mais sobre outros tipos do que imagina.

Os tipos compartilham conjuntos de operações por categorias

As strings são seqüências imutáveis: elas não podem ser alteradas no local (a parte *imutável*) e são coleções ordenadas pela posição, acessadas por meio de deslocamentos (a parte *seqüência*). Agora, acontece que todas as seqüências vistas nesta parte do livro respondem às mesmas operações de seqüência mostradas em funcionamento nas strings – concatenação, indexação, iteração etc. Mais formalmente, existem três categorias de tipo (e operação) no Python:

Números
: Suportam adição, multiplicação etc.

Seqüências
: Suportam indexação, fracionamento, concatenação etc.

Mapeamentos
: Suportam indexação pela chave etc.

Ainda não vimos os mapeamentos (os dicionários serão discutidos no próximo capítulo), mas outros tipos são parecidos, de modo geral. Por exemplo, para quaisquer objetos de seqüência `X` e `Y`:

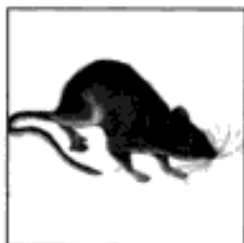
- `X + Y` gera um novo objeto seqüência com o conteúdo dos dois operandos.
- `X * N` gera um novo objeto seqüência com `N` cópias do operando `X` da seqüência.

Em outras palavras, essas operações funcionam da mesma forma, em qualquer tipo de seqüência – strings, listas, tuplas e alguns tipos de objeto definidos pelo usuário. A única diferença é que você recebe de volta um novo objeto resultado que tem o mesmo tipo dos operandos `X` e `Y` – se você concatenar listas, obterá uma nova lista e não uma string. A indexação, o fracionamento e outras operações de seqüência também funcionam da mesma forma em todas as seqüências; o tipo dos objetos que estão sendo processados indica ao Python a tarefa a ser executada.

Tipos mutáveis podem ser alterados no local

É importante conhecer a restrição da classificação de imutável, embora ela tenda a confundir os usuários iniciantes. Se um tipo de objeto é imutável, você não pode alterar seu valor no local; o Python gerará um erro, se você tentar. Em vez disso, execute código para que um novo objeto tenha um novo valor. Geralmente, os tipos imutáveis fornecem certo grau de integridade, garantindo que um objeto não seja alterado por outra parte de um programa. Você vai ver por que isso importa, quando as referências de objeto compartilhado forem discutidas, no Capítulo 7.

6

Listas de Dicionários

Este capítulo apresenta os tipos de objeto lista e dicionário – coleções de outros objetos que são os principais componentes em quase todos os scripts em Python. Conforme veremos, esses dois tipos são notavelmente flexíveis: eles podem ser alterados, podem aumentar e diminuir segundo a demanda e podem conter e serem aninhados em qualquer outro tipo de objeto. Usando bem esses tipos, podemos construir e processar estruturas de informação arbitrariamente ricas em nossos scripts.

LISTAS

A próxima parada no passeio pelos objetos internos é a *lista* do Python. As listas representam o tipo de objeto coleção ordenada mais flexível do Python. Ao contrário das strings, as listas podem conter qualquer tipo de objeto: números, strings e até outras listas. As listas do Python fazem o trabalho da maioria das estruturas de dados tipo coleção que, talvez, você tivesse que implementar manualmente em linguagens de nível mais baixo, tal como C. Em termos, algumas das propriedades principais das listas do Python são:

Coleções ordenadas de objetos arbitrários

Do ponto de vista funcional, as listas são apenas um lugar para reunir outros objetos; assim, você pode tratá-los como um grupo. As listas também definem uma ordenação posicional, da esquerda para a direita, para seus itens.

Acessadas pelo deslocamento

Exatamente como acontece com as strings, você pode buscar um objeto componente de uma lista indexando-o no deslocamento do objeto. Como nas listas os itens são ordenados pelas suas posições, você também pode executar tarefas como fracionamento e concatenação.

Têm comprimento variável, são heterogêneas e podem ser aninhadas arbitrariamente

Ao contrário das strings, as listas podem crescer e diminuir no local (elas podem ter comprimento variável) e podem conter qualquer tipo de objeto, não apenas strings de um único caractere (elas são heterogêneas). Como as listas podem conter outros objetos complexos, elas também suportam aninhamento arbitrário; você pode criar listas de listas de listas.

Categoria seqüência mutável

Em termos de nossos qualificadores de categoria de tipo, as listas podem ser alteradas no local (elas são mutáveis) e respondem a todas as operações de seqüência usadas com strings, como indexação, fracionamento e concatenação. Na verdade, as operações de seqüência funcionam da mesma forma nas listas. Como as listas são mutáveis, elas também suportam outras operações que as strings não aceitam, como exclusão e atribuição de índice.

Arrays de referências de objeto

Tecnicamente, as listas do Python contêm zero ou mais referências para outros objetos. As listas podem lembrar os arrays de ponteiros (endereços). Buscar um item de uma lista do Python é quase tão rápido quanto indexar um array da linguagem C; na verdade, as listas são realmente arrays em C dentro do interpretador do Python. O Python sempre segue uma referência para um objeto quando a referência é usada, de modo que seu programa trata apenas com objetos. Quando você insere um objeto em uma estrutura de dados ou em um nome de variável, o Python sempre armazena uma referência para o objeto e não uma cópia dele (a não ser que você solicite uma cópia explicitamente).

A Tabela 6-1 resume as operações comuns do objeto lista.

Tabela 6-1 Literais e operações comuns de lista

Operação	Interpretação
`L1 = []`	Uma lista vazia
`L2 = [0, 1, 2, 3]`	Quatro itens: índices 0..3
`L3 = ['abc', ['def', 'ghi']]`	Sublistas aninhadas
`L2[i]`	Índice,
`L3[i][j]`	índice de índice
`L2 [i:j]`	fracionamento,
`len(L2)`	comprimento
`L1 + L2`	Concatenação,
`L2 * 3`	repetição
`for x in L2`	Iteração,
`3 in L2`	participação como membro

`L2.append(4)`
`L2.extend([5,6,7])`
`L2.sort()`
`L2.index(1)`
`L2.reverse()`
	Métodos: grow, sort, search, reserve etc.

`del L2[k]`
`del L2[i:j]`
`L2.pop()`
`L2[i:j] = []`
	Redução

`L2[i] = 1`	Atribuição de índice
`L2[i:j] = [4,5,6]`	Atribuição de fracionamento
`range(4)` `xrange(0, 4)`	Produz listas/tuplas de inteiros
`L4 = [x**2 for x in range(5)]`	Abrangências de lista (Capítulo 14)

Quando escritas, as listas são codificadas como uma série de objetos (ou expressões que retornam objetos) entre colchetes, separados por vírgulas. Por exemplo, a segunda linha na

Tabela 6-1 atribui uma lista de quatro itens à variável `L2`. As listas aninhadas são codificadas como uma série entre colchetes aninhada (linha 3) e a lista vazia é apenas um par de colchetes sem nada dentro (linha 1).*

Muitas das operações da Tabela 6-1 devem parecer familiares, pois são as mesmas operações de seqüência que funcionam em strings – indexação, concatenação, iteração etc. As listas também respondem a chamadas de método específicas (que fornecem utilitários como ordenação, inversão, adição de itens no final etc.), assim como operações de alteração no local (exclusão de itens, atribuição para índices e fracionamentos etc.). As listas têm as ferramentas para operações de alteração porque representam um tipo de objeto mutável.

LISTAS EM AÇÃO

Talvez a melhor maneira de entender as listas seja vê-las em funcionamento. Vamos, mais uma vez, exemplificar algumas interações simples do interpretador, para ilustrar as operações da Tabela 6-1.

Operações de lista básicas

As listas respondem aos operadores `+` e `*` exatamente como as strings; aqui, eles também significam concatenação e repetição, exceto que o resultado é uma nova lista e não uma string. Na verdade, as listas respondem a todas as operações de seqüência gerais usadas para strings.

```
% python
>>> len([1, 2, 3])                  # Comprimento
3
>>> [1, 2, 3] + [4, 5, 6]           # Concatenação
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4                     # Repetição
['Ni!', 'Ni!', 'Ni!', 'Ni!']
>>> 3 in [1, 2, 3]                  # Participação como membro
                                      (1 significa true)
1
>>> for x in [1, 2, 3]: print x,    # Iteração
...
1 2 3
```

Falaremos mais sobre iteração com a instrução `for` e os tipos internos `range` no Capítulo 10, pois eles estão relacionados com sintaxe de instrução; em resumo, os loops `for` percorrem os itens de uma seqüência. A última entrada na Tabela 6-1, *Abrangências de lista*, será abordada no Capítulo 14; as compreensões representam uma maneira de construir listas pela aplicação de expressões a seqüências, em um único passo.

Embora + funcione de forma igual para listas e strings, é importante saber que esse operador espera o mesmo tipo de seqüência nos dois lados – caso contrário, você receberá um erro de tipo quando o código for executado. Por exemplo, você não pode concatenar uma lista e uma string, a não ser que primeiro converta a lista em uma string, usando acentos graves, `str` ou

* Na prática, você não verá muitas listas escritas dessa forma em programas de processamento de lista. É mais comum ver códigos que processam listas construídas dinamicamente (em tempo de execução). Na verdade, embora seja importante dominar a sintaxe literal, a maioria das estruturas de dados no Python é construída por meio da execução de código de programa em tempo de execução.

formatação com %. Você também pode converter a string em uma lista; a função interna list faz esse truque:

```
>>> '[1, 2]' + "34"            # O mesmo que "[1, 2]" + "34"
'[1, 2]34'
>>> [1, 2] + list ("34")       # O mesmo que [1, 2] + ["3", "4"]
[1, 2, '3', '4']
```

Indexação, fracionamento e matrizes

Como as listas são seqüências, a indexação e o fracionamento funcionam da mesma maneira, mas o resultado da indexação de uma lista depende do tipo de objeto que está no deslocamento especificado, e o fracionamento de uma lista sempre retorna uma nova lista:

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]                       # Os deslocamentos começam em zero.
'SPAM!'
>>> L[-2]                      # Negativo: conta a partir da direita.
'Spam'
>>> L[1:]                      # O fracionamento busca seções.
['Spam', 'SPAM!'
```

Uma nota aqui: como você pode aninhar listas (e outros tipos) com listas, às vezes precisará enfileirar operações de índice para ir mais fundo em uma estrutura de dados. Por exemplo, uma das maneiras mais simples de representar matrizes (arrays multidimensionais) no Python é como listas com sub-listas aninhadas. Aqui está um array bidimensional básico de 3 por 3 baseado em lista:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Com um índice, você obtém uma linha inteira (na verdade, uma sub-lista aninhada); com dois, você obtém um item dentro da linha:

```
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
>>> matrix = [   [1, 2, 3],
...              [4, 5, 6],
...              [7, 8, 9]]
>>> matrix[1][1]
5
```

Observe a última parte desse exemplo. As listas podem abranger várias linhas naturalmente, se você quiser. Posteriormente neste capítulo, você também verá uma representação de matriz baseada em dicionário. A extensão *NumPy*, mencionada anteriormente, fornece outras maneiras de manipular matrizes.

Alterando listas no local

Como as listas são mutáveis, elas suportam operações que alteram um objeto lista no local; ou seja, todas as operações desta seção modificam o objeto lista diretamente, sem obrigar o usuário a fazer uma nova cópia, como acontecia com as strings. Mas como o Python só trata

com referências de objeto, a distinção entre alterações no local e novos objeto é importante. Se você alterar um objeto no local, poderá, ao mesmo tempo, afetar mais do que uma referência para ele.

Atribuição de índice e fracionamento

Ao usar uma lista, você pode alterar seu conteúdo atribuindo a um item (deslocamento) em particular ou a uma seção inteira (fracionamento):

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'                    # Atribuição de índice
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat','more']          # Atribuição de fracionamento:
                                       exclusão+inserção
>>> L                                # Substitui os itens 0,1
['eat', 'more', 'SPAM!']
```

Tanto a atribuição de índice quanto a de fracionamento são alterações no local – elas modificam a lista de itens diretamente, em vez de gerarem um novo objeto lista como resultado. A atribuição de índice funciona como na linguagem C e na maioria das outras linguagens: o Python substitui a referência de objeto no deslocamento designado por uma nova.

A *atribuição de fracionamento*, a última operação no exemplo anterior, substitui uma seção inteira de uma lista em um único passo. Como ela pode ser um pouco complexa, é melhor considerá-la como a combinação de dois passos:

1. *Exclusão*. O fracionamento especificado à esquerda de = é excluído.
2. *Inserção*. Os novos itens à direita são inseridos na lista à esquerda, no lugar onde o fracionamento antigo foi excluído.

Não é realmente isso que acontece,* mas ajuda a esclarecer por que o número de itens inseridos não precisa corresponder ao número de itens excluídos. Por exemplo, dada uma lista `L` com o valor `[1,2,3]`, a atribuição `L[1:2]=[4,5]` configura `L` como a lista `[1,4,5,3]`. O Python primeiro exclui o número 2 (um fracionamento de um item) e depois insere os itens 4 e 5 onde estava o 2 que foi excluído. Isso também explica por que `L[1:2]=[]` é na verdade uma operação de exclusão.

Chamadas de método de lista

Assim como as strings, os objetos lista do Python também suportam chamadas de método específicas do tipo:

```
>>> L.append('please')               # Anexa chamada de método
>>> L
['eat', 'more', 'SPAM!', 'please']
>>> L.sort()                         # Ordena os itens da lista ('S' < 'e').
>>> L
['SPAM!', 'eat', 'more', 'please']
```

* Essa descrição precisa ser elaborada quando o valor e o fracionamento que estão sendo atribuídos se sobrepõem: `L[2:5]=L[3:6]`, por exemplo, funciona bem, pois o valor a ser inserido é buscado antes que a exclusão aconteça na esquerda.

Os *métodos* foram apresentados no Capítulo 5. Em resumo, eles são funções (ou atributos que referenciam funções) associadas a um objeto em particular. Os métodos fornecem ferramentas específicas do tipo; os métodos de lista apresentados aqui, por exemplo, só estão disponíveis para listas.

O método de lista `append` simplesmente anexa um item (referência de objeto) no final da lista. Ao contrário da concatenação, `append` espera que você passe um único objeto e não uma lista. O efeito de `L.append(X)` é semelhante a `L+[X]`, mas o primeiro altera `L` no local e o último cria uma nova lista.* O método `sort` ordena uma lista no local; por padrão, ele usa testes de comparação padrão do Python (aqui, comparações de string) e classifica em ordem ascendente. Você também pode passar sua própria função de comparação para `sort`.

(Tenha cuidado, pois `append` e `sort` alteram o objeto lista associado no local, mas não retornam a lista como resultado (tecnicamente, ambos retornam um valor chamado `None`). Se você escrever algo como `L=L.append(X)`, não obterá o valor modificado de `L` (na verdade, você perderá completamente a referência para a lista). Quando você usa atributos como `append` e `sort`, os objetos são alterados como um efeito colateral; portanto, não há motivo para fazer uma nova atribuição.

Assim como acontece com as strings, outros métodos de lista executam outras operações especializadas. Por exemplo, `reverse` inverte a lista no local e os métodos `extend` e `pop` inserem vários itens no final e excluem um item do final, respectivamente:

```
>>> L = [1, 2]
>>> L.extend([3,4,5])                    # Anexa vários itens.
>>> L
[1, 2, 3, 4, 5]
>>> L.pop( )                             # Exclui, retorna o último item.
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()                          # Inversão no local.
>>> L
[4, 3, 2, 1]
```

Em alguns tipos de programas, o método de lista `pop` usado aqui é freqüentemente utilizado em conjunto com `append` para implementar rapidamente uma estrutura de *pilha* tipo último a entrar, primeiro a sair. O final da lista serve como topo da pilha:

```
>>> L = []
>>> L.append(1)                          # Coloca na pilha.
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop( )                             # Retira da pilha.
2
>>> L
[1]
```

* Ao contrário da concatenação de `+`, `append` não precisa gerar novos objetos e, assim, normalmente é mais rápido. Você também pode imitar o método `append` com atribuições de fracionamento inteligentes: `L[len(L):]=[X]` é igual a `L.append(X)` e `L[:0]=[X]` é igual a anexar na frente da lista. Ambas excluem um fracionamento vazio e inserem `X`, alterando `L` no local com a mesma rapidez de `append`.

Finalmente, como as listas são mutáveis, você também pode usar a instrução `del` para excluir um item ou uma seção:

```
>>> L
['SPAM!', 'eat', 'more', 'please']
>>> del L[o]                          # Exclui um item.
>>> L
['eat', 'more', 'please']
>>> del L[1:]                         # Exclui uma seção inteira.
>>> L                                 # O mesmo que L[1:] = []
['eat']
```

Como a atribuição de fracionamento é uma exclusão mais uma inserção, você também pode excluir seções de listas, atribuindo uma lista vazia a um fracionamento (`L[i:j]=[]`). O Python exclui o fracionamento nomeado à esquerda e, em seguida, insere nada. Por outro lado, atribuir um índice a uma lista vazia apenas armazena uma referência para a lista na entrada especificada, em vez de excluí-la:

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
>>> L
['Already']
>>> L[0] = []
>>> L
[[]]
```

Aqui estão algumas indicações, antes de continuarmos. Embora todas as operações anteriores sejam típicas, existem mais métodos e operações de lista que não foram ilustrados (incluindo métodos para inserir e pesquisar). Para ver uma lista abrangente e atualizada das ferramentas de tipo, você sempre deve consultar os manuais do Python ou o livro *Python Pocket Reference* (O'Reilly) e outros textos de referência descritos no Prefácio.

Também gostaríamos de lembrá-lo, mais uma vez, que todas as operações de alteração no local anteriores só funcionam para objetos mutáveis: elas não funcionam em strings (nem em tuplas, que serão discutidas mais adiante), independente de quanto você tente.

DICIONÁRIOS

Além das listas, os *dicionários* talvez sejam o tipo de dados interno mais flexível no Python. Se você considera as listas como coleções ordenadas de objetos, os dicionários são coleções desordenadas; a principal diferença é que, nos dicionários, os itens são armazenados e buscados pela *chave*, em vez do deslocamento posicional.

Sendo um tipo interno, os dicionários podem substituir muitos dos algoritmos de pesquisa e estruturas de dados que talvez tenham que ser implementados manualmente em linguagens de nível mais baixo – indexar um dicionário é uma operação de pesquisa muito rápida. Às vezes, os dicionários fazem o trabalho de registros e tabelas de símbolos usados em outras linguagens, podem representar estruturas de dados esparsas (principalmente vazias) e muito mais. Com referência às suas principais propriedades, os dicionários são:

Acessados pela chave e não pelo deslocamento

Às vezes, os dicionários são chamados de arrays associativos ou tabelas de hashing. Eles associam um conjunto de valores a chaves, para que você possa buscar um item de um dicionário usando a chave que o armazena. Em um dicionário, você usa a mesma opera-

ção de indexação para obter componentes, mas o índice assume a forma de chave e não de um deslocamento relativo.

Coleções desordenadas de objetos arbitrários

Ao contrário das listas, os itens armazenados em um dicionário não são mantidos em uma ordem em particular; na verdade, o Python torna sua ordem aleatória para proporcionar uma pesquisa mais rápida. As chaves fornecem a localização simbólica (e não física) dos itens em um dicionário.

Têm comprimento variável, são heterogêneos e podem ser aninhados arbitrariamente

Assim como as listas, os dicionários podem aumentar e diminuir no local (sem fazer uma cópia), podem conter objetos de qualquer tipo e suportam aninhamento em qualquer profundidade (eles podem conter listas, outros dicionários etc.)

Categoria mutável mapeamento

Os dicionários podem ser alterados no local por meio da atribuição de índices, mas não suportam as operações de seqüência que funcionam em strings e listas. Como os dicionários são coleções desordenadas, operações que dependem de uma ordem fixa não têm sentido (por exemplo, concatenação, fracionamento). Em vez disso, os dicionários são os únicos representantes incorporados da categoria do tipo mapeamento – objetos que fazem o mapeamento de chaves em valores.

Tabelas de referências de objeto (tabelas de hashing)

Se as listas são arrays de referências de objeto, os dicionários são tabelas desordenadas de referências de objeto. Internamente, os dicionários são implementados como tabelas de hashing (estruturas de dados que suportam recuperação muito rápida), as quais começam pequenas e crescem segundo a demanda. Além disso, o Python emprega algoritmos de hashing otimizados para localizar chaves, de modo que a recuperação é muito rápida. Os dicionários armazenam referências de objeto (e não cópias), exatamente como as listas.

A Tabela 6-2 resume algumas das operações de dicionário mais comuns (consulte o manual da biblioteca para ver uma lista completa). Os dicionários são escritos como uma série de pares chave:valor, separados por vírgulas e incluídos entre chaves.* A mesma nota sobre a raridade relativa dos literais se aplica aqui: freqüentemente, os dicionários são construídos pela atribuição a novas chaves em tempo de execução, em vez da gravação de literais. Mas veja a seção a seguir, sobre alteração em dicionários; as listas e os dicionários crescem de maneiras diferentes. A atribuição a novas chaves funciona para dicionários, mas falha para listas (em vez disso, as listas crescem com `append`).

DICIONÁRIOS EM AÇÃO

Conforme a Tabela 6-2 sugere, os dicionários são indexados pela chave e as entradas de dicionário aninhadas são referenciadas por uma série de índices (chaves entre colchetes). Quando o Python cria um dicionário, ele armazena seus itens em qualquer ordem que escolher; para buscar um valor de volta, forneça a chave a que ele está associado. Vamos voltar ao interpretador para ter uma idéia de algumas das operações de dicionário da Tabela 6-2.

* Um dicionário vazio é um conjunto de chaves vazio; os dicionários podem ser aninhados escrevendo-se um dicionário como valor dentro de outro ou dentro de uma lista ou tupla.

Tabela 6-2 Literais e operações de dicionário comuns

Operação	Interpretação
`D1 = {}`	Dicionário vazio
`D2 = {'spam': 2, 'eggs': 3`	Dicionário de dois itens
`D3 = {'food': {'ham': 1, 'egg': 2}}`	Aninhamento
`D2['eggs']` `D3['food']['ham']`	Indexação pela chave

`D2.has_key('eggs'), 'eggs) in D2`
`D2.keys()`
`D2.values()`
`D2.copy()`
`D2.get(key, default)`
`D2.update(D1)`
	Métodos: teste de participação como membro, lista de chaves, lista de valores, cópias, padrões, intercalação etc.

`len(D1)`	Comprimento (número de entradas armazenadas)
`D2[key] = 42`	Adição/alteração
`del D2[key]`	exclusão
`D4 = dict(zip(keyslist, valslist))`	Construção (Capítulo 10)

Operações básicas de dicionário

Na operação normal, você cria dicionários, armazena e acessa itens pela chave:

```
% python
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}          # Faz um dicionário.
>>> d2['spam']                                       # Busca valor pela chave.
2
>>> d2                                               # A ordem é misturada.
{'eggs': 3, 'ham: 1, 'spam: 2}
```

Aqui, o dicionário é atribuído à variável `d2`; o valor da chave `'spam'` é o inteiro `2`. Usamos a mesma sintaxe dos colchetes para indexar dicionários pela chave, como fizemos para indexar listas pelos deslocamentos. Mas, aqui, isso significa acesso pela chave e não posição.

Observe o final desse exemplo: a ordem das chaves em um dicionário quase sempre será diferente daquela que você digitou originalmente. Isso é feito de propósito – para implementar uma pesquisa de chave rápida (conhecida como hashing), as chaves precisam se tornar aleatórias na memória. É por isso que as operações que presumem uma ordem da esquerda para a direita não se aplicam aos dicionários (por exemplo, fracionamento, concatenação); você só pode buscar valores pela chave e não pela posição.

A função interna `len` também funciona em dicionários; ela retorna o número de itens armazenados no dicionário ou, equivalentemente, o comprimento de sua lista de chaves. O método de dicionário `has_key` permite que você teste a existência de uma chave e o método `keys` retorna todas as chaves presentes no dicionário, reunidas em uma lista. Este último pode ser útil para processar dicionários em seqüência, mas você não deve depender da ordem da lista de chaves. Entretanto, como o resultado de `keys` é uma lista normal, ela sempre pode ser ordenada, se a ordem importar:

```
>>> len(d2)                    # Número de entradas no dicionário
3
>>> d2.has_key('ham')          # Teste de participação da chave como membro
                                 (1 significa true)
```

```
1
>>> 'ham' in d2              # Teste alternativo de participação da chave
                               como membro
1
>>> d2.keys()                # Cria uma nova lista de minhas chaves.
['eggs', 'ham', 'spam']
```

Observe a terceira expressão nessa listagem: o teste de participação como membro `in`, usado para strings e listas, também funciona em dicionários – ele verifica se uma chave está armazenada no dicionário, como a chamada do método `has_key` da linha anterior. Tecnicamente, isso funciona porque os dicionários definem *iteradores* que percorrem suas listas de chaves. Outros tipos fornecem iteradores que refletem seus usos comuns. Os arquivos, por exemplo, têm iteradores que lêem linha por linha. Mais informações sobre iteradores aparecem nos capítulos 14 e 21.

No Capítulo 10, você verá que a última entrada da Tabela 6-2 é outra maneira de construir dicionários por meio da passagem de listas de tuplas para a nova chamada `dict` (na verdade, um construtor de tipo), quando explorarmos a função `zip`. Trata-se de uma maneira de construir um dicionário a partir de uma chave e de listas de valores, em uma única chamada.

Alterando dicionários no local

Os dicionários são mutáveis; portanto, você pode alterá-los, expandi-los e diminuí-los no local, sem criar novos dicionários, exatamente como as listas. Basta atribuir um valor a uma chave para alterar ou criar a entrada. A instrução `del` funciona aqui também; ela exclui a entrada associada à chave especificada como índice. Observe o aninhamento de uma lista dentro de um dicionário neste exemplo (o valor de chave `'ham'`); no Python, todos os tipos de dados de coleção podem ser aninhados dentro de outros, arbitrariamente:

```
>>> d2['ham'] = ['grill', 'bake', 'fry']         # Altera entrada.
>>> d2
{'eggs': 3, 'spam': 2, 'ham': [grill', 'bake', 'fry]}
>>> del d2['eggs']                               # Exclui entrada.
>>> d2
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> d2['brunch'] = 'Bacon'                       # adiciona nova entrada.
>>> d2
{'brunch': 'bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

Assim como acontece nas listas, a atribuição a um índice existente em um dicionário altera seu valor associado. Ao contrário das listas, entretanto, ao atribuir uma *nova* chave de dicionário (que não foi atribuída antes), você cria uma nova entrada no dicionário, como foi feito no exemplo anterior para a chave `'brunch'`. Isso não funciona para listas, pois o Python considera um deslocamento fora dos limites, caso esteja além do fim de uma lista, e lança um erro. Para expandir uma lista, você precisa usar ferramentas como o método `append` ou a atribuição de fracionamento.

Mais métodos de dicionário

Além de `has_key`, os métodos de dicionário fornecem uma variedade de ferramentas. Por exemplo, os métodos de dicionário `values` e `items` retornam listas dos valores e tuplas de pares (chave,valor), respectivamente.

```
>>> d2.values(), d2.items()
([3, 1, 2], [('eggs', 3), ('ham', 1), ('spam', 2)])
```

Tais listas são úteis em loops que precisam percorrer entradas de dicionário uma a uma. Normalmente, buscar uma chave inexistente é um erro, mas o método `get` retorna um valor padrão (`None`, ou um padrão passado), caso a chave não exista.

```
>>> d2.get('spam'), d2.get('toast'), d2.get('toast', 88)
(2, None, 88)
```

O método `update` fornece aos dicionários algo semelhante à concatenação; ele intercala as chaves e os valores de um dicionário em outro, sobrescrevendo cegamente os valores da mesma chave:

```
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}
>>> d3 = {'toast':4, 'muffin':5}
>>> d2.update(d3)
>>> d2
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
```

Os dicionários também fornecem um método `copy`. Mais informações sobre esse método aparecem no próximo capítulo. Na verdade, os dicionários vêm com mais métodos do que aqueles listados na Tabela 6-2; consulte o manual da biblioteca do Python ou outras fontes de documentação para ver uma lista abrangente.

Uma tabela de linguagens

Aqui está um exemplo de dicionário mais realista. O exemplo a seguir cria uma tabela que faz o mapeamento de nomes de linguagem de programação (as chaves) para seus criadores (os valores). Você busca o nome de um criador indexando no nome da linguagem:

```
>>> table = {'Python':      'Guido van Rossum',
...           'perl':       'Larry Wall',
...           'Tcl':        'John Ousterhout' }
...
>>> language = 'Python'
>>> creator = table[language]
>>> creator
'Guido van Rossum'

>>> for lang in table.keys():
...     print lang, '\t', table[lang]
...
Tcl     John Ousterhout
Python  Guido van Rossum
Perl    Larry Wall
```

O último comando usa um loop `for`, que não abordamos ainda. Se você não estiver familiarizado com loops `for`, esse comando simplesmente faz uma iteração por cada chave presente na tabela e imprime uma lista separada por tabulações de chaves e seus valores. Consulte o Capítulo 10 para obter mais informações sobre loops `for`.

Como os dicionários não são seqüências, você não pode fazer iteração por eles diretamente com uma instrução `for`, como acontece com as strings e listas. Mas se você precisar pecorrer os itens de um dicionário, é fácil: a chamada do método de dicionário keys retorna uma lista de todas as chaves armazenadas pelas quais você faz uma iteração com um loop `for`. Se necessário, você pode indexar da chave para o valor dentro do loop `for`, como foi feito nesse código.

O Python também nos permite percorrer a lista de chaves de um dicionário sem realmente chamar o método `keys` na maioria dos loops `for`. Para qualquer dicionário `D`, escrever `for key in D:` funciona da mesma forma que escrever `for key in D.keys():`. Esse é apenas outro exemplo dos iteradores mencionados anteriormente, os quais permitem que o teste de participação como membro `in` funcione também em dicionários.

Notas sobre a utilização de dicionário

Aqui estão alguns detalhes que você deve saber quando usar dicionários:

Operações de seqüência não funcionam. Os dicionários são mapeamentos e não seqüências. Como não há nenhuma noção de ordenação entre seus itens, coisas como concatenação (uma junção ordenada) e fracionamento (extração de seção adjacente) simplesmente não se aplicam. Na verdade, se você tentar fazer tais coisas, o Python lançará um erro quando seu código for executado.

A atribuição a novos índices adiciona entradas. As chaves podem ser criadas quando você escreve um literal de dicionário (no caso em que elas são incorporadas no próprio literal) ou quando atribui valores a novas chaves de um objeto dicionário já existente. O resultado final é o mesmo.

As chaves nem sempre precisam ser strings. Nossos exemplos usaram strings como chaves, mas quaisquer outros objetos *imutáveis* (não listas) também funcionam bem. Na verdade, você poderia usar inteiros como chaves, o que faria um dicionário ser muito parecido com uma lista (ao indexar, pelo menos). Às vezes, tuplas também são usadas como chaves de dicionário, possibilitando a existência de valores de chave compostos. Além disso, objetos instância de classe (discutidos na Parte VI) também podem ser usados como chaves, desde que tenham os métodos de protocolo corretos; a grosso modo, eles precisam informar ao Python que seus valores não mudarão; senão, serão inúteis como chaves fixas.

Usando dicionários para simular listas flexíveis

Quando você usa listas, não é válido atribuir a um deslocamento que está após o fim da lista:

```
>>> L = []
>>> L[99] = 'spam'
Traceback (most recent call last):
File "<stdin>", line 1, in?
IndexError: list assignment index out of range
```

Embora você pudesse usar repetição para alocar previamente uma lista do tamanho que precisar (por exemplo, `[0]*100`), também pode fazer algo parecido com os dicionários, mas que não exige tais alocações de espaço. Usando chaves inteiras, os dicionários podem simular listas que parecem crescer em atribuição de deslocamento:

```
>>> D = {}
>>> D[99] = 'spam'
>>> D[99]
'spam'
>>> D
{99: 'spam'}
```

Aqui, parece que `D` é uma lista de 100 itens, mas na verdade é um dicionário com uma única entrada; o valor de chave `99` é a string `'spam'`. Você pode acessar essa estrutura com desloca-

mentos de forma muito parecida com as listas, mas não precisa alocar espaço para todas as posições para as quais talvez precise atribuir valores no futuro.

Usando dicionários para estruturas de dados esparsas

De maneira semelhante, as chaves de dicionário também são normalmente aprimoradas para implementar estruturas de dados *esparsas* – por exemplo, arrays multidimensionais, onde apenas algumas posições contêm valores armazenados:

```
>>> Matrix = {}
>>> Matrix[(2,3,4)] = 88
>>> Matrix[(7,8,9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4                #; separa instruções.
>>> Matrix[(X,Y,Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

Aqui, usamos um dicionário para representar um array tridimensional, onde todas as posições são vazias, exceto duas: `(2,3,4)` e `(7,8,9)`. As chaves são *tuplas* que registram as coordenadas de entradas não-vazias. Em vez de alocar uma matriz tridimensional grande e, principalmente vazia, podemos usar um dicionário simples de dois itens. Nesse esquema, acessar entradas vazias levanta uma exceção de chave inexistente – essas entradas não são armazenadas fisicamente:

```
>>> Matrix[(2,3,6)]
Traceback (most recent call last):
File "<stdin>", line 1, in?
keyError: (2, 3, 6)
```

Se quisermos preencher um valor padrão, em vez de obter uma mensagem de erro aqui, existem pelo menos três maneiras pelas quais podemos tratar desses casos. Podemos testar a existência das chaves antecipadamente em instruções `if`, usar a instrução `try` para capturar e recuperar-nos da exceção explicitamente ou simplesmente usar o método de dicionário `get`, mostrado anteriormente, para fornecer um padrão para as chaves que não existem:

```
>>> if Matrix.has_key((2,3,6)):          # Verifica a existência da chave
                                           antes de buscar.
... print Matrix[(2,3,6)]
... else:
... print 0
...
0
>>> try:
... print Matrix[(2,3,6)]                # Tenta indexar.
... except keyError:                     # Captura e se recupera.
... print 0
...
0
>>> Matrix.get((2,3,4), 0)               # Existe; busca e retorna.
88
>>> Matrix. get((2,3,6), 0)              # Não existe; usa arg padrão.
0
```

Estudaremos as instruções `if` e `try` posteriormente.

Usando dicionários como "registros"

Conforme você pode ver, os dicionários podem desempenhar muitas funções no Python. Em geral, eles podem substituir estruturas de dados de pesquisa (pois indexar pela chave é uma operação de pesquisa) e representar muitos tipos de informações estruturadas. Por exemplo, os dicionários são uma das muitas maneiras de descrever as propriedades de um item no domínio de seu programa; eles podem ter a mesma função de "registros" ou "estruturas" em outras linguagens:

```
>>> rec = {}
>>> rec['name'] = 'mel'
>>> rec['age'] = 41
>>> rec['job'] = 'trainer/writer'
>>>
>>> print rec['name']
mel
```

Esse exemplo preenche o dicionário fazendo atribuições a novas chaves, com o passar do tempo. Especialmente quando aninhados, os tipos de dados internos do Python nos permitem representar informações estruturadas facilmente:

```
>>> mel = {'name': 'Mark',
...        'jobs': ['trainer/writer'],
...        'web': 'www.rmi.net/~lutz',
...        'home': {'state': 'CO', 'zip':80501}}
```

Esse exemplo usa um dicionário para capturar propriedades de objeto novamente, mas codifica tudo de uma vez (em vez de atribuir a cada chave separadamente) e aninha uma lista e um dicionário para representar valores de propriedade da estrutura. Para buscar componentes de objetos aninhados, basta usar operações de indexação juntas:

```
>>> mel['name']
'Mark'
>>> mel['jobs']
['trainer', writer']
>>> mel['jobs'][1]
'writer'
>>> mel ['home']['zip']
80501
```

Finalmente, note que mais maneiras de construir dicionários podem surgir com o passar do tempo. No Python 2.3, por exemplo, as chamadas `dict(name='mel', age=41"` e `dict([('name', 'bob'), ('age',30)])` também constroem dicionários de duas chaves. Consulte os capítulos 10, 13 e 27 para ver mais detalhes.

Por que isto é relevante: interfaces de dicionário

Além de ser uma forma de armazenar informações pela chave em seus programas, algumas extensões do Python também apresentam interfaces que se parecem e funcionam como dicionários. Por exemplo, a interface do Python para arquivos dbm de acesso pela chave é muito parecida com um dicionário que deve ser aberto; as strings são armazenadas e buscadas usando-se índices de chave:

```
import anydbm
file = anydbm.open("filename")          # Vincula a file.
file['key'] = 'data'                    # Armazena data pela chave.
data = file['key']                      # Busca data pela chave.
```

Posteriormente, você verá que também podemos armazenar objetos Python inteiros dessa maneira, se substituirmos `anydbm` por `shelve` (shelves são bancos de dados de acesso pela chave de objetos persistentes do Python). Para trabalho na Internet, o suporte para script CGI também apresenta uma interface tipo dicionário; uma chamada para `cgi.FieldStorage` produz um objeto tipo dicionário, com uma entrada por campo de entrada na página da Web do cliente:

```
import cgi
form = cgi.FieldStorage()               # Analisa dados de form.
if form.has_key('name'):
    showReply('Hello, ' + form['name'].value)
```

Tudo isso (e os dicionários) são casos de mapeamentos. Mais informações sobre scripts CGI aparecerão posteriormente neste livro.

7

Tuplas, Arquivos e Tudo Mais

Este capítulo encerra nosso exame dos tipos de objeto básicos do Python, apresentando a *tupla* (uma coleção de outros objetos que não pode ser alterada) e o *arquivo* (uma interface para arquivos externos em seu computador). Conforme você verá, a tupla é um objeto relativamente simples que, de modo geral, executa operações sobre as quais você já aprendeu quando estudamos as strings e as listas. O objeto arquivo é uma ferramenta completa e comumente usada para processar arquivos. Mais exemplos de arquivo aparecerão em capítulos posteriores deste livro.

Este capítulo também conclui esta parte do livro examinando as propriedades comuns a todos os tipos de dados básicos que conhecemos – as noções de igualdade, comparações, cópias de objeto etc. Também exploraremos brevemente outros tipos de objeto das ferramentas do Python. Conforme veremos, embora tenhamos conhecido todos os tipos internos primários, a história dos objetos no Python é mais ampla do que sugerimos até aqui. Finalmente, encerraremos esta parte com um conjunto de armadilhas comuns dos tipos de dados, e com exercícios que permitirão experimentar as idéias que você aprendeu.

TUPLAS

O último tipo de coleção em nosso levantamento é a tupla do Python. As tuplas constróem grupos de objetos simples. Elas funcionam exatamente como as listas, exceto que não podem ser alteradas no local (são imutáveis) e, normalmente, são escritas como uma série de itens entre parênteses e não entre colchetes. Embora não suportem nenhuma chamada de método, as tuplas compartilham a maioria das suas propriedades com as listas. As tuplas:

São coleções ordenadas de objetos arbitrários

Assim como as strings e as listas, as tuplas são uma coleção de objetos ordenados pela posição; assim como as listas, elas podem incorporar qualquer tipo de objeto.

São acessadas pelo deslocamento

Assim como nas strings e nas listas, os itens em uma tupla são acessados pelo deslocamento (e não pela chave). As tuplas suportam todas as operações de acesso baseadas em deslocamento, como indexação e fracionamento.

São da categoria seqüência imutável

Assim como as strings, as tuplas são imutáveis; elas não suportam nenhuma das operações de alteração no local aplicadas às listas. Assim como as strings e listas, as tuplas são seqüências; elas suportam muitas das mesmas operações.

Têm comprimento fixo, são heterogêneas e podem ser aninhadas arbitrariamente

Como as tuplas são imutáveis, elas não podem aumentar nem diminuir sem criar uma nova tupla. Por outro lado, as tuplas podem conter outros objetos compostos (por exemplo, listas, dicionários, outras tuplas) e, assim, suportam aninhamento arbitrário.

São arrays de referências de objeto

Assim como as listas, as tuplas são melhor consideradas como arrays de referência de objeto; as tuplas armazenam pontos de acesso para outros objetos (referências), e indexar uma tupla é relativamente rápido.

A Tabela 7-1 destaca as operações de tupla comuns. As tuplas são escritas como uma série de objetos (na realidade, expressões que geram objetos), separados por vírgulas e incluídos entre parênteses. Uma tupla vazia é apenas um par de parênteses sem nada dentro.

Tabela 7-1 Operações e literais de tupla comuns

Operação	Interpretação
`()`	Uma tupla vazia
`t1 = (0,)`	Uma tupla de um item (não é uma expressão)
`t2 = (0, 'Ni', 1.2, 3)`	Uma tupla de quatro itens
`t2 = 0, 'Ni', 1.2, 3`	Outra tupla de quatro itens (igual a linha anterior)
`t3 = ('abc', ('def', 'ghi'))`	Tuplas aninhadas
`t1[i]`	Índice,
`t3[i][j]`	índice de índice
`t1[i:j]`	fracionamento,
`len(t1)`	comprimento
`t1 + t2`	Concatenação,
`t2 * 3`	repetição
`for x in t2`	Iteração,
`3 in t2`	participação como membro

Note que as tuplas não têm métodos (por exemplo, uma chamada de `append` não funcionará aqui), mas suportam as operações de seqüência comuns que vimos para as strings e listas:

```
>>> (1, 2) + (3, 4)                    # Concatenação
(1, 2, 3, 4)
>>> (1, 2) * 4                         # Repetição
(1, 2, 1, 2, 1, 2, 1, 2)
>>> T = (1, 2, 3, 4)                   # Indexação, fracionamento
>>> T[0], T[1:3]
(1, (2, 3))
```

A segunda e a quarta entradas da Tabela 7-1 merecem um pouco mais de explicação. Como os parênteses também podem englobar expressões (veja a seção "Números", no Capítulo 4), você precisa fazer algo especial para informar ao Python quando um único objeto entre parênteses é uma tupla e não uma expressão simples. Se você realmente quiser uma tupla

de um único item, basta adicionar uma vírgula no final, após o item e antes do parênteses de fechamento:

```
>>> x = (40)              # Um inteiro
>>> x
40
>>> y (40,)               # Uma tupla contendo um inteiro
>>> y
(40,)
```

Como um caso especial, o Python também permite que você omita os parênteses de abertura e fechamento de uma tupla, em contextos onde não é sintaticamente ambíguo fazer isso. Por exemplo, a quarta linha da tabela simplesmente listou quatro itens, separados por vírgulas. No contexto de uma instrução de atribuição, o Python reconhece isso como uma tupla, mesmo não tendo parênteses. Para os iniciantes, o melhor conselho é que, provavelmente, é mais fácil usar parênteses do que descobrir quando eles são opcionais. Muitos programadores também acham que os parênteses tendem a ajudar na legibilidade do script.

Fora as diferenças da sintaxe de literal, as operações de tupla (as três últimas linhas na Tabela 7-1) são idênticas às strings e listas. As únicas diferenças que devemos prestar atenção, são que as operações +, * e de fracionamento retornam novas *tuplas*, quando aplicadas às tuplas, e que estas não fornecem os métodos que você viu para strings, listas e dicionários. Se você quiser ordenar uma tupla, por exemplo, normalmente precisará primeiro convertê-la para uma lista, para ter acesso a uma chamada de método de ordenação e transformá-la em um objeto mutável:

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)
>>> tmp.sort()
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)
>>> T
('aa', 'bb', 'cc', 'dd')
```

Aqui, as funções internas `list` e `tuple` foram usadas para converter para uma lista e, depois, de volta para uma tupla; na realidade, as duas chamadas criam novos objetos, mas o resultado é igual a uma conversão. Note também que a regra sobre a imutabilidade da tupla só se aplica ao nível superior da própria tupla e não ao seu conteúdo; uma lista dentro de uma tupla, por exemplo, pode ser alterada normalmente:

```
>>> T = (1, [2, 3], 4)
>>> T[1][0] = 'spam'                # Funciona
>>> T
(1, ['spam', 3], 4)
>>> T[1] = 'spam'                   # Falha
TypeError: object doesn't support item assignment
```

Por que listas e tuplas?

Essa parece ser a primeira pergunta que sempre surge ao se ensinar sobre tuplas para iniciantes: por que precisamos de tuplas, se temos as listas? Parte disso pode ser por motivos históricos. Mas a melhor resposta parece ser que a imutabilidade das tuplas proporciona alguma *integridade* – você pode ter certeza de que uma tupla não mudará por meio de outra referência, em outra parte do programa. Não há essa garantia para as listas.

As tuplas também podem ser usadas em lugares que as listas não podem – por exemplo, como *chaves* de dicionário (veja o exemplo de matriz esparsa no Capítulo 6). Algumas operações internas também podem exigir ou sugerir tuplas e não listas. Como regra geral, as listas são a ferramenta de escolha para coleções ordenadas que talvez precisem mudar; as tuplas tratam dos outros casos.

ARQUIVOS

A maioria dos leitores provavelmente estão familiarizados com a noção de *arquivos* – compartimentos de armazenamento nomeados em seu computador, gerenciados pelo seu sistema operacional. Este último tipo de objeto interno fornece uma maneira de acessar esses arquivos dentro de programas em Python. A função interna `open` cria um objeto arquivo do Python, o qual serve como um vínculo para um arquivo residente em sua máquina. Após chamar `open`, você pode ler e gravar o arquivo externo associado, chamando métodos do objeto arquivo. O nome interno `file` é sinônimo de `open` e os arquivos podem ser abertos chamando-se qualquer um desses nomes.

Comparados com os tipos que você viu até aqui, os objetos arquivo são um tanto incomuns. Eles não são números, seqüências nem mapeamentos; em vez disso, eles exportam métodos apenas para tarefas comuns de processamento de arquivo.

A Tabela 7-2 resume as operações comuns de arquivo. Para abrir um arquivo, um programa chama a função `open`, com o nome externo primeiro, seguido de um modo de processamento (`'r'` para abrir para entrada – o padrão – `'w'` para criar e abrir para saída, `'a'` para abrir para anexar no final e outros que omitiremos aqui). Os dois argumentos devem ser strings do Python. O argumento do nome do arquivo externo pode incluir um prefixo de caminho de diretório absoluto ou relativo e específico da plataforma; sem um caminho, supõe-se que o arquivo existe no diretório de trabalho corrente (isto é, onde o script é executado).

Tabela 7-2 Operações e literais de tupla comuns

Operação	Interpretação
`output = open('/tmp/spam', 'w')`	Cria arquivo de saída (`'w'` significa gravação).
`input = open('data', 'r')`	Cria arquivo de entrada (`'r'` significa leitura).
`S = input.read()`	Lê o arquivo inteiro em uma única string.
`S = input.read(N)`	Lê N bytes (1 ou mais).
`S = input.readline()`	Lê a próxima linha (até o marcador de final de linha)
`L = input.readlines()`	Lê o arquivo inteiro na lista de strings da linha.
`output.write(S)`	Grava a string `S` no arquivo.
`output.writelines(L)`	Grava no arquivo todas as strings da linha da lista `L`.
`output.close()`	Fechamento manual (feito para você quando o arquivo é coletado).

Uma vez que você tenha um objeto arquivo, chame seus métodos para ler ou gravar no arquivo externo. Em todos os casos, o texto do arquivo assume a forma de strings nos programas em Python; ler um arquivo retorna seu texto em strings e texto é passado como strings para os métodos de gravação. Tanto a leitura como a gravação têm várias modalidades. A Tabela 7-2 fornece as mais comuns.

Chamar o método de arquivo `close` termina sua conexão com o arquivo externo. No Python, o espaço na memória de um objeto é recuperado automaticamente assim que o objeto não

é mais referenciado em nenhuma parte do programa. Quando os objetos arquivo são recuperados, o Python também fecha o arquivo automaticamente, se necessário. Por isso, você não precisa fechar sempre seus arquivos manualmente, especialmente em scripts simples que não são executados por muito tempo. Por outro lado, as chamadas de fechamento manual não causam danos e normalmente são uma boa idéia em sistemas maiores. Rigorosamente falando, esse recurso de fechamento automático dos arquivos em coleção não faz parte da definição da linguagem e pode mudar com o passar do tempo. Por isso, é bom habituar-se com as chamadas manuais do método de arquivo `close`.

Arquivos em ação

Aqui está um exemplo simples que demonstra os fundamentos do processamento de arquivos. Ele primeiro abre um novo arquivo para saída, grava uma string (terminada com um marcador de nova linha, `'\n'`) e fecha o arquivo. Posteriormente, o exemplo abre o mesmo arquivo novamente, no modo de entrada, e lê a linha de volta. Note que a segunda chamada de `readline` retorna uma string vazia; é assim que os métodos de arquivo do Python informam que você chegou ao final do arquivo (as linhas vazias no arquivo voltam como strings apenas com um caractere de nova linha e não como strings vazias).

```
>>> myfile = open ('myfile', 'w')            # Abre para saída (cria).
>>> myfile.write('hello text file\n')        # Grava uma linha de texto.
>>> myfile.close()

>>> myfile = open('myfile', 'r')             # Abre para entrada.
>>> myfile.readline()                        # Lê a linha de volta.
'hello text file\n'
>>> myfile.readline()                        # String vazia: fim do arquivo
''
```

Existem outros métodos de arquivo mais avançados, não mostrados na Tabela 7-2. Por exemplo, `seek` reconfigura sua posição corrente em um arquivo (a próxima leitura ou gravação acontece na posição), `flush` obriga a saída do buffer a ser gravada no disco (por padrão, os arquivos são sempre colocados em buffer) etc.

O quadro "Por que isto é relevante: varredores de arquivo", no Capítulo 10, esboça os padrões de código de loop comuns para percorrer arquivos, e os exemplos de partes posteriores deste livro discutem um código maior, baseado em arquivo. Além disso, o manual da biblioteca padrão do Python e os livros de referência descritos no Prefácio fornecem uma lista completa dos métodos de arquivo.

CATEGORIAS DE TIPO REVISTAS

Agora que já vimos todos os tipos internos básicos do Python, vamos dar uma olhada em algumas das propriedades que eles compartilham.

A Tabela 7-3 classifica todos os tipos que vimos, de acordo com as categorias de tipo apresentadas anteriormente. Os objetos compartilham operações de acordo com sua categoria – por exemplo, strings, listas e tuplas, todas compartilham operações de seqüência. Apenas os objetos mutáveis podem ser alterados no local. Você pode alterar listas e dicionários no local, mas não números, strings ou tuplas. Os arquivos só exportam métodos; portanto, a mutabilidade não se aplica realmente (eles podem ser alterados ao serem gravados, mas isso não é o mesmo que as restrições de tipo do Python).

Tabela 7-3 Classificações de objeto

Tipo de objeto	Categoria	Mutável?
Números	Numérico	Não
Strings	Seqüência	Não
Listas	Seqüência	Sim
Dicionários	Mapeamento	Sim
Tupla	Seqüência	Não
Arquivos	Extensão	n/a

Por que isto é relevante: sobrecarga de operador

Posteriormente, veremos que os objetos que implementamos com classes podem selecionar e escolher uma dessas categorias arbitrariamente. Por exemplo, se você quiser fornecer um novo tipo de objeto seqüência especializado, que seja consistente com as seqüências internas, desenvolva uma classe que sobrecarregue coisas como indexação e concatenação:

```
class MySequence:
    def __getitem__(self, index):
        # Chamado em self[index], outros
    def __add__(self, other):
        # Chamado em self + other
```

etc. Você também pode tornar o novo objeto mutável ou não, implementando seletivamente métodos chamados por operações de alteração no local (por exemplo, `__setitem__` é chamado em atribuições `self[indice]=valor`). Também é possível implementar novos objetos em C, como tipos de extensão C. Para eles, você preencherá entradas de ponteiro de função C para escolher entre conjuntos de operação numérica, de seqüência e mapeamento.

GENERALIDADE DE OBJETO

Vimos vários tipos de objeto compostos (coleções com componentes). Em geral:

- Listas, dicionários e tuplas podem conter qualquer tipo de objeto.
- Listas, dicionários e tuplas podem ser aninhados arbitrariamente.
- Listas e dicionários podem aumentar e diminuir dinamicamente.

Como suportam estruturas arbitrárias, os tipos de objeto compostos do Python são bons na representação de informações complexas em um programa. Por exemplo, os valores nos dicionários podem ser listas, as quais podem conter tuplas, dicionários e assim por diante – com a profundidade de aninhamento que for necessária para modelar os dados a serem processados.

Aqui está um exemplo de aninhamento. A interação a seguir define uma árvore de objetos seqüência compostos aninhados, mostrada na Figura 7-1. Para acessar seus componentes, você pode incluir tantas operações de índice quantas forem exigidas. O Python avalia os índices da esquerda para a direita e, em cada passo, busca uma referência para um objeto mais profundamente aninhado. A Figura 7-1 pode ser uma estrutura de dados patologicamente complicada, mas ilustra a sintaxe usada para acessar objetos aninhados em geral:

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1]
[(1, 2), ([3], 4)]
>>> L[1][1]
```

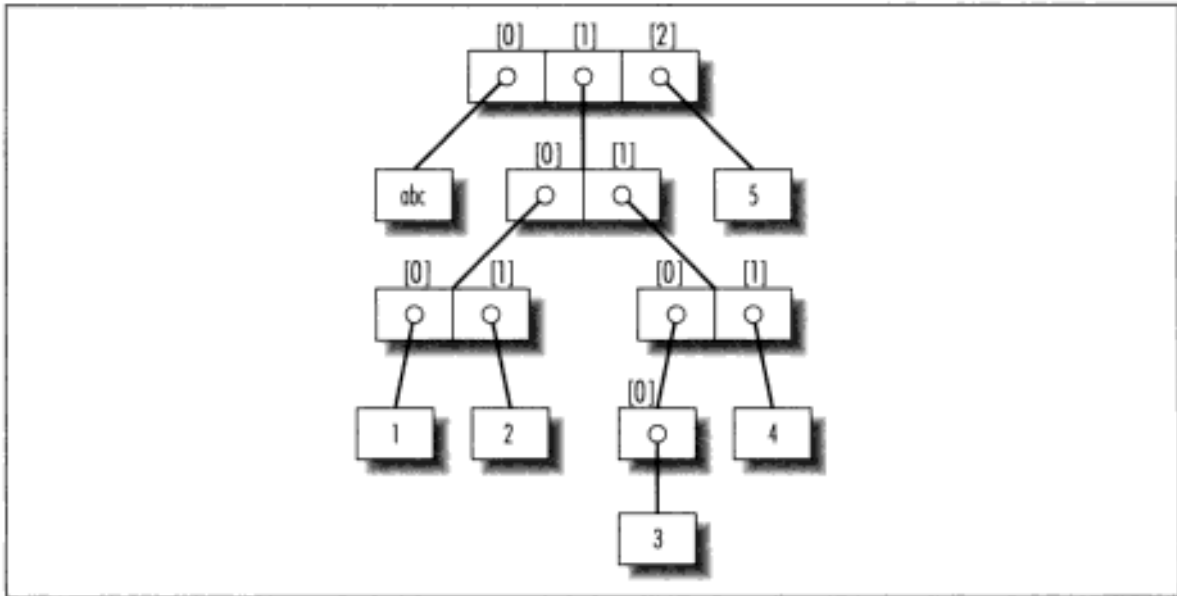



Figura 7-1 Uma árvore de objetos aninhados.

```
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```

REFERÊNCIAS *VERSUS* CÓPIAS

A seção "O entreato da tipagem dinâmica", no Capítulo 4, mencionou que as atribuições sempre armazenam referências para objetos e não cópias. Na prática, normalmente é isso que você quer. Mas, como as atribuições podem gerar várias referências para o mesmo objeto, às vezes você precisa estar ciente de que alterar um objeto mutável no local pode afetar outras referências para o mesmo objeto em outra parte de seu programa. Se você não quiser tal comportamento, precisará instruir o Python para que copie o objeto explicitamente.

Assim, o exemplo a seguir cria uma lista atribuída a `X` e outra atribuída a `L` que incorpora uma referência para a lista `X`. Ele também cria um dicionário `D`, contendo outra referência para a lista `X`:

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']            # Incorpora referências para o objeto de X.
>>> D = {'x':X, 'y':2}
```

Neste ponto, existem três referências para a primeira lista criada: a partir do nome `X`, dentro da lista atribuída a `L` e dentro do dicionário atribuído a `D`. A situação está ilustrada na Figura 7-2.

Como as listas são mutáveis, alterar o objeto lista compartilhada, a partir de qualquer uma das três referências, altera o que as outras duas referenciam:

```
>>> X[1] = 'surprise'            # Altera todas as três referências!
>>> L
['a', [1, 'surprise', 3], 'b']
>>> D
{'x': [1, 'surprise', 3], 'y': 2}
```

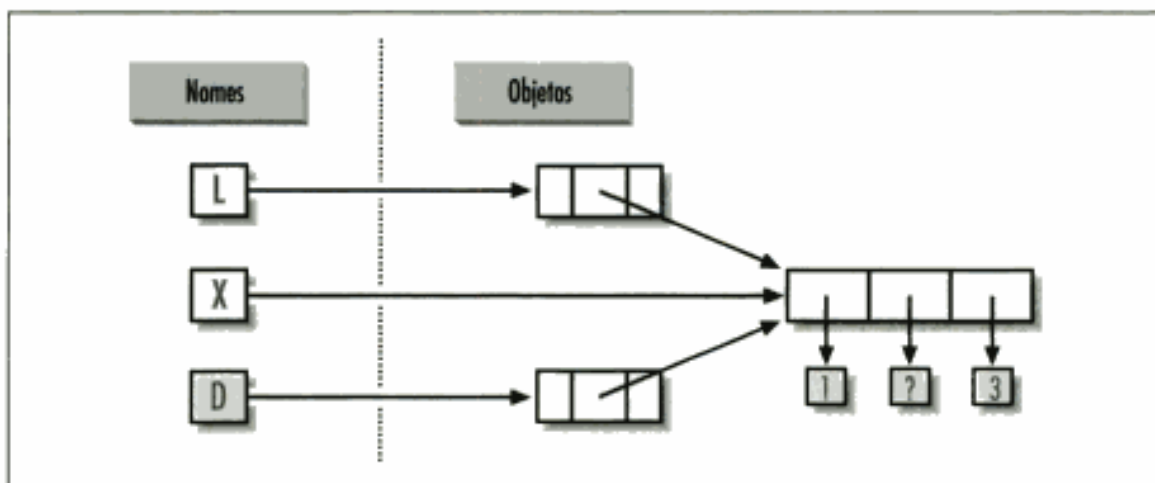



Figura 7-2 Referências de objeto compartilhadas.

As referências são uma analogia de nível mais alto aos ponteiros de outras linguagens. Embora você não possa pegar a referência em si, é possível armazenar a mesma referência em mais de um lugar: em variáveis, em listas etc. Esse é um recurso – você pode passar um objeto grande em um programa, sem gerar cópias dele pelo caminho. Se você realmente quer cópias, pode solicitá-las:

- Expressões de fracionamento com limites vazios copiam seqüências.
- O método de dicionário `copy` copia um dicionário.
- Algumas funções internas, como `list`, também fazem cópias.
- O módulo de biblioteca padrão `copy` faz cópias completas.

Por exemplo, se você tiver uma lista e um dicionário e não quiser que seus valores sejam alterados por meio de outras variáveis:

```
>>> L = [1,2,3]
>>> D = {'a':1, 'b':2}
```

simplesmente atribua cópias a outras variáveis e não referências para os mesmos objetos:

```
>>> A[1] = L[:]                # Em vez de: A = L (ou list(L))
>>> B = D.copy()               # Em vez de B = D
```

Desse modo, as alterações feitas a partir de outras variáveis alteram as cópias e não os originais:

```
>>> A[1] = 'Ni'
>>> B['c'] = 'spam'
>>>
>>> L, D
([1, 2, 3], {'a': 1, 'b': 2})
>>> A, B
([1, 'Ni', 3], {'a': 1, 'c': 'spam', 'b': 2})
```

Em termos do exemplo original, você pode evitar os efeitos colaterais da referência fracionando a lista original, em vez de simplesmente atribuir um nome a ela:

```
>>> X = [1, 2, 3]
>>> L = ['a', X[:], 'b']           # Incorpora cópias do objeto de X.
>>> D = {'x':X[:], 'y':2}
```

Isso muda o quadro da Figura 7-2 – `L` e `D` apontarão para listas diferentes de `X`. O resultado é que as alterações feitas por meio de `X` terão impacto apenas em `X` e não em `L` e `D`. Analogamente, as alterações feitas em `L` ou `D` não terão impacto em `X`.

Uma nota sobre cópias: os fracionamentos de limite vazio e o método `copy` de dicionários ainda só fazem uma cópia *de nível superior* – eles não copiam estruturas de dados aninhadas, se alguma estiver presente. Se você precisar de uma cópia completa, totalmente independente, de uma estrutura de dados profundamente aninhada, use o módulo de cópia padrão `import copy` e, digamos, `X=copy.deepcopy(Y)` para copiar totalmente um objeto `Y` arbitrariamente aninhado. Essa chamada percorre os objetos recursivamente para copiar todas as suas partes. Contudo, esse é um caso muito mais raro (e é o motivo pelo qual você precisa escrever mais para fazer funcionar). Normalmente, as referências são o comportamento que você desejará; quando não forem, os métodos de fracionamento e cópia normalmente serão a cópia que precisará fazer.

COMPARAÇÕES, IGUALDADE E VERDADE

Todos os objetos do Python também respondem às comparações: testes de igualdade, grandeza relativa etc. As comparações do Python sempre inspecionam todas as partes de objetos compostos, até que um resultado possa ser determinado. Na verdade, quando objetos aninhados estão presentes, o Python percorre as estruturas de dados automaticamente para aplicar comparações *recursivamente* – da esquerda para a direita e com a profundidade que for necessária.

Por exemplo, uma comparação de objetos lista compara todos os seus componentes automaticamente:

```
>>> L1 = [1, ('a', 3)]              # O mesmo valor, objetos exclusivos
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2              # Equivalente? O mesmo objeto?
(1, 0)
```

Aqui, listas equivalentes são atribuídas a `L1` e `L2`, mas são atribuídos objetos distintos. Por causa da natureza das referências do Python (estudadas no Capítulo 4), existem duas maneiras de testar a igualdade:

O operador `==` testa equivalência de valor. O Python efetua um teste de equivalência, comparando todos os objetos aninhados recursivamente.

O operador `is` testa a identidade do objeto. O Python testa se os dois são realmente o mesmo objeto (isto é, estão no mesmo endereço na memória).

No exemplo, `L1` e `L2` passam no teste `==` (elas têm valores equivalentes, pois todos os seus componentes são equivalentes), mas falham na verificação `is` (elas são dois objetos diferentes e, assim, são duas partes diferentes da memória). Observe o que acontece para strings curtas:

```
>>> S1 = 'spam'
>>>S2 = 'spam'
>>> S1 == S2, S1 is S2
(1, 1)
```

Aqui, devemos ter dois objetos distintos que, por acaso, têm o mesmo valor: `==` deve ser verdadeiro e `is` deve ser falso. Como internamente o Python coloca strings curtas no cache e as reutiliza, como uma ação de otimização, na verdade existe apenas uma string, 'spam',

na memória, compartilhada por S1 e S2. Assim, o teste de identidade `is` relata um resultado verdadeiro. Para ativar o comportamento normal, precisamos usar strings mais longas, que ficam fora do mecanismo de cache:

```
>>> S1 = 'a longer string'
>>> S2 = 'a longer string'
>>> S1 == S2, S1 is S2
(1, 0)
```

Como as strings são imutáveis, o mecanismo de colocação de objeto no cache é irrelevante para seu código – a string não pode ser alterada no local, independente de quantas variáveis se refiram a ela. Se os testes de identidade parecem confusos, consulte a seção "O entreato da tipagem dinâmica", no Capítulo 4, para recordar os conceitos de referência de objeto.

Como regra geral, o operador `==` é o que você desejará usar para quase todas as verificações de igualdade; `is` é reservado para funções altamente especializadas. Veremos casos dos dois operadores em uso, posteriormente no livro.

Note que as comparações de grandeza relativa também são aplicadas recursivamente em estruturas de dados aninhadas:

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2        # Menor,igual,maior: tupla de
                                        resultados
(0, 0, 1)
```

Aqui, `L1` é maior que `L2` porque o `3` aninhado é maior do que `2`. O resultado da última linha acima é, na verdade, uma tupla de três objetos – os resultados das três expressões digitadas (um exemplo de tupla sem parênteses de inclusão).

Os três valores nessa tupla representam valores verdadeiros e falsos; no Python, um valor inteiro `0` representa falso e um valor inteiro `1` representa verdadeiro. O Python também reconhece qualquer estrutura de dados vazia como um valor falso e qualquer estrutura de dados não vazia como um valor verdadeiro. Mais geralmente, as noções de verdadeiro e falso são propriedades intrínsecas de todo objeto no Python – cada objeto é verdadeiro ou falso, como segue:

- Os números são verdadeiros, se forem diferentes de zero.
- Os outros objetos são verdadeiros, se não estiverem vazios.

A Tabela 7-4 dá exemplos de objetos verdadeiros e falsos no Python.

Tabela 7-4 Exemplos de valores verdadeiros e falsos de objetos

Objeto	Valor
`"spam"`	Verdadeiro
`""`	Falso
`[]`	Falso
`{}`	Falso
`1`	Verdadeiro
`0.0`	Falso
`None`	Falso

O Python também fornece um objeto especial chamado `None` (o último item na Tabela 7-4), que também é considerado falso. `None` é o único valor de um tipo de dados especial no Python; normalmente, ele serve como um lugar reservado vazio, muito parecido com um ponteiro NULL na linguagem C.

Por exemplo, lembre-se de que, para listas, você não pode atribuir a um deslocamento, a não ser que ele já exista (a lista não aumenta de forma mágica, se você fizer uma atribuição fora dos limites). Para alocar previamente uma lista de 100 itens, para que possa adicionar a qualquer um dos 100 deslocamentos, você pode preencher uma lista com objetos `None`:

```
>>> L = [None] * 100
>>>
>>> L
[None, None, None, None, None, None, None, ...]
```

O novo tipo booleano na versão 2.3

O Python 2.3 introduz um novo tipo de dados booleano explícito, chamado `bool`, com valores `True` e `False` disponíveis como novos nomes internos atribuídos previamente. Devido à maneira como esse novo tipo é implementado, na verdade trata-se apenas de uma extensão secundária nas noções de verdadeiro e falso delineadas neste capítulo, projetada para tornar valores verdadeiro e falso mais explícitos. De qualquer modo, a maioria dos programadores estava atribuindo previamente `True` e `False` a `1` e `0`; portanto, o novo tipo torna isso um padrão. Por exemplo, um loop infinito agora pode ser codificado como `while True:`, em vez do menos intuitivo `while 1:`. Analogamente, flags podem ser inicializados com `flag = False`.

Internamente, os novos nomes `True` e `False` são instâncias de `bool`, que, por sua vez, é apenas uma subclasse do tipo de dados inteiro interno `int`. `True` e `False` se comportam exatamente como os inteiros `1` e `0`, exceto que têm lógica de impressão personalizada – eles são impressos como as palavras `True` e `False`, em vez dos algarismos `1` e `0` (tecnicamente, `bool` redefine seus formatos de string `str` e `repr`). Por causa dessa personalização, a partir do Python 2.3, a saída de expressões booleanas digitadas no prompt interativo imprime as palavras `True` e `False`, em vez dos valores `1` e `0` que você vê neste livro.

Para todos os outros propósitos práticos, você pode tratar `True` e `False` como se fossem variáveis predefinidas configuradas com os valores inteiros `1` e `0` (por exemplo, `True + 3` produz `4`). Nos testes de verificação de verdade, `True` e `False` são avaliadas como `true` e `false`, pois são apenas versões especializadas dos inteiros `1` e `0`. Além disso, você não é obrigado a usar apenas tipos booleanos em instruções `if`; todos os objetos ainda são inerentemente verdadeiros ou falsos e todos os conceitos booleanos mencionados neste capítulo ainda funcionam como antes. Mais informações sobre valores booleanos aparecerão no Capítulo 9.

Em geral, o Python compara os tipos como segue:

- Os números são comparados pela grandeza relativa.
- As strings são comparadas lexicograficamente, caractere por caractere ("abc" < "ac").
- As listas e tuplas são comparadas pela comparação de cada componente, da esquerda para a direita.
- Os dicionários são comparados como a comparação de listas ordenadas (chave, valor).

Em capítulos posteriores, veremos outros tipos de objeto que podem mudar a maneira como são comparados.

HIERARQUIAS DE TIPO DO PYTHON

A Figura 7-3 resume todos os tipos de objeto internos disponíveis no Python e seus relacionamentos. Vimos os mais importantes deles; a maioria dos outros tipos de objetos da Figura 7-3 corresponde a unidades de programa (por exemplo, funções e módulos) ou ao funcionamento interno exposto do interpretador (por exemplo, quadros de pilha e código compilado).

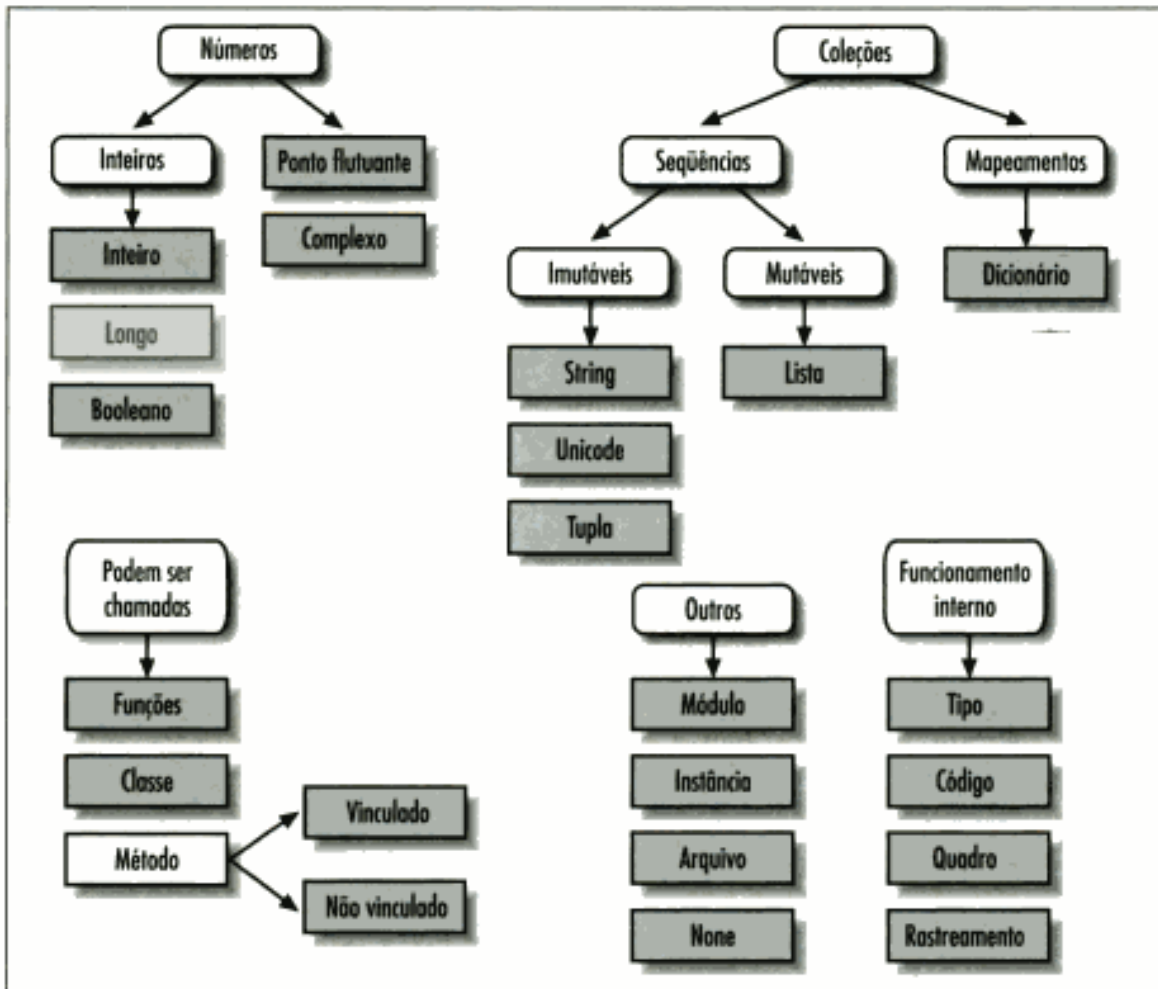



Figura 7-3 Hierarquias de tipo internas.

O principal ponto a notar aqui é que tudo é um tipo de objeto em um sistema Python e pode ser processado por seus programas em Python. Por exemplo, você pode passar uma classe para uma função, atribuí-la a uma variável, preenchê-la em uma lista ou dicionário etc.

Até os tipos são um tipo de objeto no Python: uma chamada para a função interna `type(X)` retorna o objeto tipo do objeto `X`. Os objetos de tipo podem ser usados para comparações de tipo manuais em instruções `if` do Python. Entretanto, por motivos a serem explicados na Parte IV, normalmente os testes manuais de tipo não são a coisa certa a fazer no Python.

Uma nota sobre nomes de tipo: a partir do Python 2.2, cada tipo básico tem um novo nome interno adicionado para suportar subclasses de tipo: `dict`, `list`, `str`, `tuple`, `int`, `long`, `float`, `complex`, `unicode`, `type` e `file` (`file` é sinônimo de `open`). As chamadas para esses nomes são, na verdade, chamadas de construtor de objeto e não simplesmente funções de conversão.

O módulo `types` fornece mais nomes de tipo (agora, de modo geral, sinônimos dos nomes de tipo internos) e é possível fazer testes de tipo com a função `isinstance`. Por exemplo, na versão 2.2, todos os testes de tipo a seguir são verdadeiros:

```
isinstance([1], list)
type([1])==list
type([1])==type([])
type([1])==types.ListType
```

Como os tipos podem ser subclasses na versão 2.2, a técnica de `isinstance` geralmente é recomendada. Consulte o Capítulo 23 para ver mais informações sobre subclasse de tipos internos na versão 2.2 e posteriores.

OUTROS TIPOS NO PYTHON

Além dos objetos básicos que estudamos neste capítulo, uma instalação típica do Python tem dezenas de outros tipos de objeto disponíveis como extensões C ou classes Python vinculadas. Você verá exemplos de alguns deles posteriormente no livro – objetos expressão regular, arquivos DBM, componentes de janela de GUI etc. A principal diferença entre essas ferramentas extras e os tipos internos que acabamos de ver é que os tipos internos fornecem sintaxe de criação de linguagem especial para seus objetos (por exemplo, 4 para um inteiro, `[1,2]` para uma lista, a função `open` para arquivos). Outras ferramentas geralmente são exportadas em um módulo interno, que você precisa primeiro importar para usar. Consulte a referência da biblioteca do Python para ver um guia abrangente de todas as ferramentas disponíveis para programas em Python.

PROBLEMAS DOS TIPOS INTERNOS

A Parte II termina com uma discussão sobre os problemas comuns que parecem atrapalhar os usuários iniciantes (e os especialistas ocasionais), junto com suas soluções.

A atribuição cria referências e não cópias

Como esse é um conceito muito importante, é mencionado novamente: você precisa entender o que está acontecendo com as referências compartilhadas em seu programa. Por exemplo, a seguir, o objeto lista atribuído ao nome `L` é referenciado a partir de `L` e dentro da lista atribuída ao nome `M`. Alterar `L` no local altera também o que `M` referencia:

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']              # Incorpora uma referência para L.
>>> M
['X', [1, 2, 3], 'Y']
>>> L[1] = 0                       # Altera M também
>>> M
['X', [1, 0, 3], 'Y']
```

Normalmente, esse efeito se torna importante apenas em programas maiores e as referências compartilhadas freqüentemente são exatamente o que você quer. Se não forem, você pode evitar o compartilhamento de objetos copiando-os explicitamente; para listas, você sempre pode fazer uma cópia de nível superior usando um fracionamento de limites vazios:

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']            # Incorpora uma cópia de L.
>>> L[1] = 0                        # Altera apenas L e não M
```

```
>>> L
[1, 0, 3]
>>> M
['X', [1, 2, 3], 'Y']
```

Lembre-se de que os limites do fracionamento têm como padrão `0` e o comprimento da seqüência que está sendo fracionada. Se ambos forem omitidos, o fracionamento extrairá todo item presente na seqüência e, assim, fará uma cópia de nível superior (um novo objeto não compartilhado).

A repetição acrescenta um nível de profundidade

Repetição de seqüência é como adicionar uma seqüência nela mesma várias vezes. Isso é verdade, mas quando seqüências mutáveis são aninhadas, o efeito pode nem sempre ser o que você espera. Por exemplo, no código a seguir, `L` é atribuído a `X` quatro vezes repetidas, enquanto `Y` recebe uma lista *contendo* `L` quatro vezes repetidas:

```
>>> L = [4, 5, 6]
>>> X = L *4                    # É como [1, 5, 6] + [4, 5, 6] + ...
>>> Y = [L] * 4                 # [L] +[L] + ... = [L, L,...]

>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Como `L` foi aninhada na segunda repetição, `Y` acaba incorporando referências para a lista original atribuída a `L`, e está aberta aos mesmos tipos de efeitos colaterais observados na última seção:

```
>>> L[1] = 0                    # Tem impacto em Y mas não em X
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

Soluções

Na verdade, essa é outra maneira de criar o caso da referência de objeto mutável; portanto, as mesmas soluções anteriores se aplicam aqui. E se você se lembrar que a repetição, a concatenação e o fracionamento copiam apenas o nível superior de seus objetos operando, esses tipos de casos fazem muito mais sentido.

Estruturas de dados cíclicas

Na verdade, encontramos esse problema em um exercício anterior: se um objeto coleção contém uma referência para ele mesmo, ele é chamado de objeto cíclico. O Python imprime `"[...]"` quando detecta um ciclo no objeto, em vez de ficar travado em um loop infinito:

```
>>> L = ['grail']              # Anexa referência para o mesmo objeto.
>>> L.append(L)                # Gera ciclo no objeto: [...]
>>> L
['grail', [...]]
```

Além de entender que os três pontos representam um ciclo no objeto, é importante conhecer esse caso em geral, pois ele pode levar a problemas – as estruturas cíclicas podem fazer com

que o código entre em loops inesperados, se você não os antecipar. Por exemplo, alguns programas mantêm uma lista ou dicionário de itens já visitados e fazem uma verificação para saber se encontraram um ciclo. Veja as soluções dos exercícios da Parte I no Apêndice B para obter mais informações sobre o problema, e o programa `reloadall.py`, no final do Capítulo 18, para encontrar uma solução.

Não use uma referência cíclica, a não ser que precise. Existem bons motivos para se criar ciclos, mas, a não ser que você tenha código que saiba como tratar deles, provavelmente na prática não desejará fazer seus objetos referenciarem eles mesmos com muita freqüência.

Os tipos imutáveis não podem ser alterados no local

Finalmente, você não pode alterar um objeto imutável no local:

```
T = (1, 2, 3)
T[2] = 4                              # Erro!
T= T[:2] + (4,)                       # Tudo bem: (1, 2, 4)
```

Construa um novo objeto com fracionamento, concatenação etc., e atribua-o de volta à referência original, se necessário. Isso pode parecer trabalho de desenvolvimento extra, mas o resultado é que os problemas anteriores não acontecem ao se usar objetos imutáveis como tuplas e strings; como eles não podem ser alterados no local, não estão abertos aos tipos de efeitos colaterais para os quais as listas estão.

EXERCÍCIOS DA PARTE II

Esta sessão o convida a entrar nos fundamentos dos objetos internos. Como antes, algumas idéias novas podem aparecer pelo caminho; portanto, consulte o Apêndice B, quando tiver terminado (e mesmo quando não tiver). Se você tiver pouco tempo, sugerimos começar com o exercício 11 (o mais prático deles) e depois trabalhar do primeiro até o último, quando o tempo permitir. Contudo, este material inteiro é fundamental; portanto, tente fazer o máximo que puder.

1. *Os fundamentos*. Experimente interativamente as operações de tipo comuns encontradas nas tabelas da Parte II. Para começar, ative o prompt interativo do Python, digite cada uma das expressões a seguir e tente explicar o que está acontecendo em cada caso:

```
2 ** 16
2 / 5, 2 / 5.0
"spam" + "eggs"
S = "ham"
"eggs " + S
S * 5
S[:0]
"green %s and %s" % ("eggs", S)

('x',)[0]
('x', 'y')[1]

L = [1,2,3] + [4,5,6]
L, L[:], L[:0], L[-2], L[-2:]
([1,2,3] + [4,5,6])[2:4]
[L[2], L[3]]
L.reverse(); L
```

```
L.sort(); L
L.index(4)

{'a':1, 'b':2}['b']
D = {'x':1, 'y':2, 'z':3}
D['w'] = 0
D['x'] + D['w']
D[(1,2,3)]] = 4
D.keys(), D.values(), D.has_key((1,2,3))

[[]], ["",[],(),{},None]
```

2. *Indexação e fracionamento*. No prompt interativo, defina uma lista chamada `L` que contenha quatro strings ou números (por exemplo, `L=[0,1,2,3]`). Em seguida, experimente alguns casos de limite.
   a. O que acontece quando você tenta indexar fora dos limites (por exemplo, `L[4]`)?
   b. E quanto ao fracionamento fora dos limites (por exemplo, `L[-1000:100]`)?
   c. Finalmente, como o Python trata disso, se você tentar extrair uma seqüência na ordem inversa – com o limite inferior maior do que o limite superior (por exemplo, `L[3:1]`)? Dica: tente atribuir a este fracionamento (`L[3:1]=['?']`) e veja onde o valor é colocado. Você acha que esse pode ser o mesmo fenômeno que viu ao fracionar fora dos limites?
3. *Indexação, fracionamento e del*. Defina novamente outra lista `L` com quatro itens e atribua uma lista vazia a um de seus deslocamentos (por exemplo, `L[2]=[]`). O que acontece? Em seguida, atribua uma lista vazia a um fracionamento (`L[2:3]=[]`). O que acontece agora? Lembre-se de que a atribuição de fracionamento exclui e insere o novo valor onde ele estava. A instrução `del` exclui deslocamentos, chaves, atributos e nomes. Utilize-a em sua lista para excluir um item (por exemplo, `del L[0]`). O que acontece se você exclui um fracionamento inteiro (`del L[1:]`)? O que acontece quando você atribui algo que não é uma seqüência a um fracionamento (`L[1:2]=1`)?
4. *Atribuição de tupla*. Digite a seguinte seqüência:

   ```
   >>> X = 'spam'
   >>> Y = 'eggs'
   >>> X, Y = Y, X
   ```

   O que você acha que está acontecendo com `X` e `Y` quando digita essa seqüência?
5. *Chaves de dicionário*. Considere os trechos de código a seguir:

   ```
   >>> D = {}
   >>> D[1] = 'a'
   >>> D[2] = 'b'
   ```

   Aprendemos que os dicionários não são acessados por deslocamentos. Então, o que está acontecendo aqui? O seguinte joga alguma luz sobre o assunto? (Dica: strings, inteiros e tuplas compartilham qual categoria de tipo?)

   ```
   >>> D[(1, 2, 3)] = 'c'
   >>> D
   {1: 'a', 2: 'b', (1, 2, 3): 'c'}
   ```
6. *Indexação de dicionário*. Crie um dicionário chamado `D`, com três entradas para as chaves `'a'`, `'b'` e `'c'`. O que acontece se você tenta indexar uma chave inexistente (`D['d']`)? O que o Python faz se você tenta atribuir a uma chave `d` inexistente (por

exemplo, `D['d']='spam'`)? Como isso se compara com as atribuições e referências fora dos limites para listas? Isso parece uma regra para nomes de variável?

7. *Operações genéricas.* Faça testes interativos para responder as questões a seguir:
   a. O que acontece quando você tenta usar o operador + em tipos diferentes/misturados (por exemplo, string + lista, lista + tupla)?
   b. O operador + funciona quando um dos operandos é um dicionário?
   c. O método `append` funciona para listas e para strings? E quanto ao uso do método `keys` em listas? (Dica: o que `append` presume sobre seu objeto?)
   d. Finalmente, que tipo de objeto você recebe de volta quando fraciona ou concatena duas listas ou duas strings?
8. *Indexação de string.* Defina uma string `S` de quatro caracteres: `S = "spam"`. Em seguida, digite a seguinte expressão: `S[0][0][0][0][0]`. Alguma pista sobre o que está acontecendo desta vez? (Dica: lembre-se de que uma string é uma coleção de caracteres, mas os caracteres do Python são strings de um único caractere.) Essa expressão de indexação ainda funcionará se você a aplicar em uma lista como `['s', 'p', 'a', 'm']`? Por quê?
9. *Tipos imutáveis.* Defina uma string `S` de `4` caracteres, novamente: `S = "spam"`. Escreva uma atribuição que altere a string para `"slam"`, usando apenas fracionamento e concatenação. Você poderia executar a mesma operação usando apenas indexação e concatenação? E quanto a atribuição de índice?
10. *Aninhamento.* Escreva uma estrutura de dados que represente suas informações pessoais: nome (primeiro, do meio, último), idade, ocupação, endereço, endereço de email e número de telefone. Você pode construir a estrutura de dados com qualquer combinação de tipos de objeto internos que quiser: listas, tuplas, dicionários, strings, números. Em seguida, acesse os componentes individuais de suas estruturas de dados por meio de indexação. Algumas estruturas fazem mais sentido do que as outras para esse objeto?
11. *Arquivos.* Escreva um script que crie um novo arquivo de saída chamado *myfile.txt* e escreva nele a string `"Hello file world!"`. Em seguida, escreva outro script que abra *myfile.txt*, leia e imprima seu conteúdo. Execute seus dois scripts a partir da linha de comando do sistema. O novo arquivo aparece no diretório onde você executou seus scripts? E se você adicionar um caminho de diretório diferente no nome de arquivo passado para `open`? Nota: os métodos de arquivo `write` não adicionam caracteres de nova linha em suas strings; adicione um `'\n'` explícito no final da string, se você quiser terminar completamente a linha no arquivo.
12. *A função dir revisitada.* Tente digitar as expressões a seguir no prompt interativo. A partir da versão 1.5, a função dir foi generalizada para listar todos os atributos de qualquer objeto do Python em que você provavelmente estará interessado. Se você estiver usando uma versão anterior à 1.5, o esquema __methods__ tem o mesmo efeito. Se você estiver usando o Python 2.2, a função `dir` provavelmente será a única dessas que funcionará.

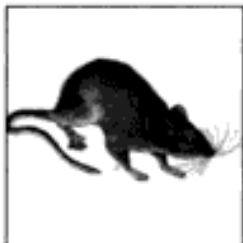
```
[].__methods__              # 1.4 ou 1.5
dir([])                     # 1.5 e posteriores

{}.__methods__              # Dicionário
dir({})
```

Instruções e Sintaxe

Na Parte III, estudaremos o conjunto de instruções procedurais do Python: instruções que selecionam ações alternativas, repetem operações, imprimem objetos etc. Como essa é a primeira vez que vemos as instruções formalmente, também exploraremos o modelo de sintaxe geral do Python. Conforme veremos, o Python tem um modelo de sintaxe simples e familiar, embora freqüentemente digitemos muito menos nas instruções dessa linguagem do que em algumas outras.

Também conheceremos as expressões booleanas, em conjunto com instruções condicionais e loops, e aprenderemos sobre os esquemas de documentação do Python, enquanto estudamos a sintaxe das strings e comentários de documentação. Em um nível abstrato, as instruções que conheceremos aqui são usadas para criar e processar os objetos da Parte II. Ao chegar ao final desta parte, você poderá escrever e executar programas com lógica em Python.

8

Atribuição, Expressões e Impressão

Agora que já vimos os tipos de objeto internos básicos do Python, este capítulo explora suas formas de instruções fundamentais. Em termos simples, as instruções são as coisas que você escreve para dizer ao Python o que seus programas devem fazer. Se os programas fazem *coisas* com *algo*, as instruções são a maneira pela qual você especifica que tipo de *coisas* um programa faz. O Python é uma linguagem procedural, baseada em instruções; combinando instruções, você especifica um procedimento que o Python executa para satisfazer os objetivos de um programa.

Outra maneira de entender a função das instruções é rever a hierarquia conceitual apresentada no Capítulo 4, na qual falamos sobre objetos internos e as expressões usadas para manipulá-los. Este capítulo eleva a hierarquia para o próximo nível:

1. Os programas são compostos de módulos.
2. Os módulos contêm instruções.
3. As instruções contêm expressões.
4. As expressões criam e processam objetos.

Em sua essência, a sintaxe do Python é composta de instruções e expressões. As expressões processam objetos e são incorporadas em instruções. As instruções desenvolvem a *lógica* mais ampla da operação de um programa – elas usam e controlam as expressões para processar os objetos que já vimos. Além disso, é nas instruções que os objetos começam a existir (por exemplo, em expressões dentro de instruções de atribuição) e algumas instruções criam tipos de objetos inteiramente novos (funções, classes etc.). Sempre existem instruções em módulos, os quais são gerenciados com instruções.

A Tabela 8-1 resume o conjunto de instruções do Python. A Parte III trata das entradas na tabela através de `break` e `continue`. Você será apresentado informalmente a algumas das instruções da Tabela 8-1. A Parte III completará os detalhes que pulamos anteriormente, apresentará o restante do conjunto de instruções procedurais do Python e abordará o modelo da sintaxe global.

Tabela 8-1 Instruções do Python

Instrução	Função	Exemplo
Atribuição	Criar referências	`curly, moe, larry = 'good', 'bad', 'ugly'`
Chamadas	Executar funções	`stdout.write("spam, ham, toast\n")`
`print`	Imprimir objetos	`print 'The Killer', joke`
`if/elif/else`	Selecionar ações	`if "python" in text: print text`
`for/else`	Iteração de seqüência	`for x in mylist: print x`
`while/else`	Loops gerais	`while 1: print 'hello'`
`pass`	Lugar reservado vazio	`while 1: pass`
`break, continue`	Saltos de loop	`while 1:` `   if not line: break`
`try/except/finally`	Captura de exceções	`try: action()` `except: print 'action error'`
`raise`	Ativar exceção	`raise endSearch, location`
`import, from`	Acesso a módulo	`import sys; from sys import stdin`
`def, return, yield`	Construção de funções	`def f(a, b, c=1, *d): return a+b+c+d[0]` `def gen(n): for i in n, yield i*2`
`class`	Construção de objetos	`class subclass: staticData = []`
`global`	Espaços de nome	`def function(): global x, y; x = 'new'`
`del`	Exclusão de referências	`del data[k]; del data[i:j]; del obj.attr`
`exec`	Execução de strings de código	`exec "import" + modName in gdict, ldict`
`assert`	Verificações de depuração	`assert X > Y`

As instruções relacionadas a unidades de programa maiores – funções, classes, módulos e exceções – levam a idéias de programação mais amplas, de modo que cada uma delas terá uma seção própria. As instruções mais exóticas, como `exec` (que compila e executa código construído como strings), serão abordadas posteriormente no livro ou podem ser encontradas na documentação padrão do Python.

INSTRUÇÕES DE ATRIBUIÇÃO

Já usamos a instrução de atribuição do Python para atribuir objetos a nomes. Em sua forma básica, você escreve um *destino* de uma atribuição à esquerda de um sinal de igualdade e um *objeto* a ser atribuído, à direita. O destino à esquerda pode ser um nome ou componente de objeto e o objeto à direita pode ser uma expressão arbitrária que calcula um objeto. De modo geral, a atribuição é simples de usar, mas aqui estão algumas propriedades a serem lembradas:

As atribuições criam referências de objeto. A atribuição do Python armazena referências para objetos em nomes ou entradas de estrutura de dados. Ela sempre cria referências para objetos, em vez de copiar objetos. Por isso, as variáveis do Python são muito mais parecidas com ponteiros do que com áreas de armazenamento de dados.

Os nomes são criados ao serem atribuídos pela primeira vez. O Python cria nomes de variável na primeira vez que você atribui um valor (uma referência de objeto) para eles. Não há necessidade de declarar nomes previamente. Algumas entradas de estrutura de dados (mas não todas) também são criadas ao serem atribuídas (por exemplo, entradas de dicionário, alguns atributos de objeto). Uma vez atribuído, um nome é substituído pelo valor que referencia, quando aparece em uma expressão.

Os nomes devem ser atribuídos antes de serem referenciados. Inversamente, é um erro usar um nome para o qual você ainda não atribuiu um valor. O Python lançará uma exceção se você tentar, em vez de retornar algum tipo de valor padrão ambíguo (e difícil de notar).

Atribuições implícitas:* `import`, `from`, `def`, `class`, `for`, *argumentos de função. Nesta seção, estamos preocupados com a instrução `=`, mas a atribuição ocorre em muitos contextos no Python. Por exemplo, veremos posteriormente que as importações de módulo, definições de função e de classe, variáveis de loop `for` e argumentos de função, são todas atribuições implícitas. Como a atribuição funciona da mesma forma onde quer que apareça, todos esses contextos simplesmente vinculam nomes a referências de objeto em tempo de execução.

A Tabela 8-2 ilustra as diferentes instruções de atribuição do Python. Além dessa tabela, o Python contém um conjunto de formas de instrução de atribuição conhecido como *atribuição ampliada*.

Tabela 8-2 Formas de instrução de atribuição

Operação	Interpretação
`spam ='Spam'`	Forma básica
`Spam, ham = 'yum', 'YUM'`	Atribuição de tupla (posicional)
`[spam, ham] = ['yum', 'YUM']`	Atribuição de lista (posicional)
`spam = ham = 'lunch'`	Destino múltiplo

A primeira linha na Tabela 8-2 é, de longe, a mais comum: vincular um único objeto a um nome (ou entrada de estrutura de dados). As outras entradas da tabela representam formas especiais:

Atribuições de desempacotamento de tupla e lista

A segunda e a terceira linhas são relacionadas. Quando você escreve tuplas ou listas no lado esquerdo do `=`, o Python combina objetos no lado direito com destinos no lado esquerdo e os atribui da esquerda para a direita. Por exemplo, na segunda linha da tabela, o nome `spam` é atribuído à string `'yum'` e o nome `ham` é vinculado à string `'YUM'`. Internamente, o Python primeiro faz uma tupla dos itens à direita, de modo que isso é freqüentemente chamado de atribuição de desempacotamento de tupla (e lista).

Atribuições com destino múltiplo

A última linha na Tabela 8-2 mostra a forma de atribuição com destino múltiplo. Nessa forma, o Python atribui uma referência ao mesmo objeto (o objeto que está mais à direita) para todos os destinos à esquerda. Na tabela, os nomes `spam` e `ham` receberiam ambos uma referência para o mesmo objeto string `'lunch'` e, portanto, compartilhariam a mesma referência para o objeto. O efeito é o mesmo que se você tivesse codificado `ham='lunch'`, seguido de `spam=ham`, pois `ham` é avaliada como o objeto string original.

Já usamos a atribuição básica. Aqui estão alguns exemplos simples de atribuição de desempacotamento em ação:

```
% python
>>> nudge = 1
>>> wink = 2
>>> A, B = nudge, wink                # Atribuição de tupla
>>> A, B                              # Igual a A = Nudge; B = wink
```

```
(1, 2)
>>> [C, D] = [nudge,wink]            # Atribuição de lista
>>> C, D
(1, 2)
```

A atribuição de tupla leva a um truque de desenvolvimento comum em Python, que foi apresentado em uma solução para os exercícios da Parte II. Como o Python cria uma tupla temporária que salva os valores da direita, as atribuições de desempacotamento também são uma maneira de *trocar* o valor de duas variáveis sem criar sua própria tupla temporária:

```
>>> nudge = 1
>>> wink = 2
>>> nudge, wink = wink, nudge        # Tuplas: troca valores
>>> nudge, wink                      # Igual a T = nudge; nudge = wink; wink
= T
(2, 1)
```

As formas de atribuição de tupla e lista são generalizadas para aceitar qualquer tipo de seqüência à direita, desde que tenha o mesmo comprimento. Você pode atribuir uma tupla de valores a uma lista de variáveis, uma string de caracteres a uma tupla de variáveis etc. Em todos os casos, o Python atribui itens na seqüência da direita às variáveis na seqüência da esquerda pela posição, da esquerda para a direita:

```
>>> [a, b, c] = (1, 2, 3)
>>> a, c
(1, 3)
>>> (a,b,c) = "ABC"
>>> a, c
('A', 'C')
```

A atribuição de desempacotamento também origina outro idioma de desenvolvimento comum em Python: atribuir uma série de inteiros a um conjunto de variáveis:

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
```

Isso inicializa os três nomes com os códigos inteiros `0`, `1` e `2` respectivamente (é o equivalente no Python aos tipos de dados *enumerados* que talvez você tenha visto em outras linguagens). Para entender isso, você também precisa saber que a função interna `range` gera uma lista de inteiros sucessivos:

```
>>> range(3)
[0, 1, 2]
```

Como `range` é comumente usada para loops `for`, falaremos mais sobre ela no Capítulo 10.

Regras de nome de variável

Agora que já vimos as instruções de atribuição, é hora de sermos mais formais no uso de nomes de variável. No Python, os nomes começam a existir quando você atribui valores a eles, mas existem algumas regras a seguir ao escolher nomes para coisas em seu programa:

Sintaxe: (sublinhado ou letra) + (qualquer número de letras, algarismos ou sublinhados)

Os nomes de variável devem começar com um sublinhado ou com uma letra, seguido de qualquer número de letras, algarismos ou sublinhados. `_spam`, `spam` e `Spam_1` são nomes válidos, mas `1_Spam`, `spam$` e `@#!`, não.

Maiúsculo importa: SPAM não é o mesmo que spam

O Python sempre presta atenção a maiúsculos nos programas, tanto nos nomes que você cria como em palavras reservadas. Por exemplo, os nomes `X` e `x` referem-se a duas variáveis diferentes.

As palavras reservadas estão fora dos limites

Os nomes que definimos não podem ser iguais às palavras que têm significado especial na linguagem Python. Por exemplo, se tentarmos usar um nome de variável como `class`, o Python acusará um erro de sintaxe, mas `klass` e `Class` funcionam bem. A Tabela 8-3 lista as palavras reservadas (e portanto, fora dos limites) no Python.

Tabela 8-3 Palavras reservadas do Python

`and`	`del`	`for`	`is`	`raise`
`assert`	`elif`	`from`	`lambda`	`return`
`break`	`else`	`global`	`not`	`try`
`class`	`except`	`if`	`or`	`while`
`continue`	`exec`	`import`	`pass`	`yield`*
`def`	`finally`	`in`	`print`	

* `yield` é uma extensão opcional na versão 2.2, mas é uma palavra-chave padrão na versão 2.3. Ela é usada em conjunto com funções geradoras, um recurso mais recente discutido no Capítulo 14.

As palavras reservadas do Python são sempre todas em letras minúsculas. E elas são realmente reservadas; ao contrário dos nomes no escopo interno, que você conhecerá na próxima parte, não é possível redefinir palavras reservadas por meio de atribuição (por exemplo, `and=1` é um erro de sintaxe).* Além disso, como os nomes de módulo em instruções de importação se tornam variáveis em seu script, essa restrição se estende aos seus nomes de arquivo de módulo – você pode escrever um arquivo chamado *and.py*, mas não pode importá-lo. Vamos rever essa idéia na Parte V.

Convenções de atribuição de nomes

Além dessas regras, existe também um conjunto de *convenções* de atribuição de nomes – regras que não são obrigatórias, mas são normalmente usadas na prática. Por exemplo, como nomes com dois sublinhados no início e no fim (por exemplo, `__nome__`) geralmente têm significado especial para o interpretador do Python, você deve evitar esse padrão para seus próprios nomes. Aqui está uma lista de todas as convenções seguidas pelo Python:

- Os nomes que começam com um único sublinhado (`_X`) não são importados por uma instrução `from module import *` (descrita no Capítulo 16).
- Os nomes que têm dois sublinhados no início e no fim (`__X__`) são nomes definidos pelo sistema, os quais têm significado especial para o interpretador.
- Os nomes que começam com dois sublinhados e não terminam com mais dois (`__X`) são localizados ("mutilados") nas classes que os englobam (descrito no Capítulo 23).
- O nome que é apenas um único sublinhado (`_`) mantém o resultado da última expressão, ao se trabalhar interativamente.

* Contudo, na implementação do Python baseada em Java, o Jython, às vezes as variáveis definidas pelo usuário podem ser iguais às palavras reservadas.

Além dessas convenções do interpretador do Python, conheceremos outras que os programadores de linguagem normalmente também seguem. Na Parte VI, por exemplo, veremos que é comum os nomes de classe começarem com uma letra maiúscula e que o nome `self`, embora não seja reservado, normalmente tem uma função especial. E, na Parte IV, estudaremos outra classe de nomes, conhecida como *internos*, os quais são predefinidos, mas não reservados (e, portanto, podem ser reatribuídos: `open=42` funciona, embora você pudesse desejar que não funcionasse).

Os nomes não têm tipo, mas os objetos têm

É fundamental manter clara a distinção do Python entre nomes e objetos. Conforme descrito na seção "O entreato da tipagem dinâmica", no Capítulo 4, os objetos têm um tipo (por exemplo, inteiro, lista) e podem ser mutáveis ou não. Por outro lado, os nomes (também conhecidos como variáveis) são sempre apenas referências para objetos; eles não têm nenhuma noção de mutabilidade e nenhuma informação de tipo associada, fora o tipo do objeto que por acaso referenciam em dado momento.

Na verdade, é perfeitamente correto atribuir o mesmo nome a diferentes tipos de objetos, em diferentes momentos:

```
>>> x = 0                    # x é vinculado a um objeto inteiro
>>> x = "Hello"              # Agora, ele é uma string.
>>> x = [1, 2, 3]            # E, agora, é uma lista.
```

Em exemplos posteriores, você verá que essa natureza genérica dos nomes pode ser uma vantagem decisiva na programação em Python.* Na Parte IV, você aprenderá que os nomes também vivem em algo chamado *escopo*, que define onde eles podem ser usados; o lugar em que você atribui um nome determina onde ele é visível.

Instruções de atribuição ampliadas

A partir do Python 2.0, está disponível um conjunto de formatos de instrução de atribuição adicionais, listados na Tabela 8-4. Conhecidos como *atribuição ampliada* e emprestados da linguagem C, esses formatos são principalmente apenas atalhos. Eles envolvem a combinação de uma expressão binária e uma atribuição. Por exemplo, os dois formatos a seguir agora são praticamente equivalentes:

```
X = X + Y                    # Forma tradicional
X += Y                       # Forma ampliada mais recente
```

Tabela 8-4 Instalações de atribuição ampliadas

`X += Y`	`X &= Y`	`X -= Y`	`X |= Y`
`X *= Y`	`X ^= Y`	`X /= Y`	`X >>= Y`
`X %= Y`	`X <<= Y`	`X **= Y`	`X //= Y`

A atribuição ampliada funciona em qualquer tipo que suporte a expressão binária indicada. Por exemplo, aqui estão duas maneiras de somar `1` a um nome:

```
>>> x = 1
>>> x = x + 1                          # Tradicional
>>> x
```

* Se você já usou a linguagem C, pode estar interessado em saber que no Python não há nenhuma noção da declaração `const` da linguagem C++. Certos objetos podem ser imutáveis, mas os nomes sempre podem ser atribuídos. O Python também tem maneiras de ocultar nomes em classes e módulos, mas eles não são iguais às declarações da linguagem C++.

```
2
>>> x += 1                              # Ampliada
>>> x
3
```

Quando aplicada a uma string, a forma ampliada realiza uma concatenação – exatamente como se você tivesse digitado a forma mais longa: `S = S + "SPAM"`:

```
>>> S = "spam"
>>> S += "SPAM"                         # Concatenação implícita
>>> S
'spamSPAM'
```

Conforme mostrado na Tabela 8-4, existem duas formas de atribuição ampliada análogas, para cada operador de expressão binária do Python (um operador com valores no lado esquerdo e no lado direito). Por exemplo, `X*=Y` multiplica e atribui, `X>>=Y` desloca para a direita e atribui e assim por diante. `X //= Y` (para divisão na base) é novidade na versão 2.2. As atribuições ampliadas têm três vantagens:*

Há menos coisas para você digitar. Precisamos dizer mais?

Elas só precisam avaliar o lado esquerdo uma vez. Em `X+=Y`, `X` poderia ser uma expressão de objeto complicada. Na forma ampliada, ela só precisa ser avaliada uma vez. Na forma longa, `X=X+Y`, `X` aparece duas vezes e deve ser executada duas vezes.

Elas escolhem a técnica otimizada automaticamente. Para objetos que suportam alterações no local, as formas ampliadas executam essas operações automaticamente, em vez de fazerem cópias mais lentas.

O último ponto aqui exige um pouco mais de explicação. Para atribuições ampliadas, as operações no local podem ser aplicadas em objetos mutáveis como uma otimização. Lembre-se de que as listas podem ser estendidas de várias maneiras. Para adicionar um item no final de uma lista, podemos concatenar ou usar `append`:

```
>>> L = [1, 2]
>>> L = L + [3]                         # Concatenação: mais lenta
>>> L
[1, 2, 3]
>>> L.append(4)                         # Mais rápida, mas no local
>>> L
[1, 2, 3, 4]
```

E para adicionar um conjunto de itens no final, podemos concatenar novamente ou chamar o método de lista `extend`:**

```
>>> L = L + [5, 6]                      # Concatenação: mais lenta
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8])                    # Mais rápida, mas no local
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

* Programadores de C/C++, tomem nota: embora agora o Python tenha coisas como `X+=Y`, ele ainda não tem operadores de incremento/decremento automático da linguagem C (por exemplo, `X++`, `--X`). Eles não são bem mapeados no modelo de objeto do Python, pois não há nenhuma noção de alteração no local para objetos imutáveis, como números.

** Conforme sugerido no Capítulo 6, também podemos usar atribuição de fracionamento: `L[len(L):] = [11,12,13]`, mas isso funciona de forma aproximadamente igual ao método de lista `extend`, mais simples.

Em todos esses exemplos, a concatenação é menos propensa aos efeitos colaterais das referências de objeto compartilhadas, mas geralmente será executada mais lentamente do que a equivalente no local. A concatenação precisa criar um novo objeto, copiar na lista da esquerda e depois copiar na lista da direita. Em contraste, as chamadas de método no local simplesmente adicionam itens no final de um bloco de memória.

Quando a atribuição ampliada é usada para estender uma lista, podemos nos esquecer desses detalhes – o Python chama automaticamente o método mais rápido `extend`, em vez da operação de concatenação mais lenta indicada por +:

```
>>> L += [9, 10]                          # Mapeado para L.extend([9, 10])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

INSTRUÇÕES DE EXPRESSÃO

No Python, você também pode usar expressões como instruções. Mas como o resultado da expressão não será salvo, só tem sentido fazer isso se a expressão fizer algo útil como efeito colateral. Normalmente, as expressões são usadas como instruções em duas situações:

Para chamadas de funções e métodos

Algumas funções e métodos fazem muito trabalho sem retornar um valor. Como você não está interessado em manter o valor que elas retornam, pode chamar essas funções com uma instrução de expressão. Às vezes, essas funções são chamadas de `procedures`, em algumas linguagens. No Python, elas assumem a forma de funções que não retornam valor.

Para imprimir valores no prompt interativo

O Python ecoa os resultados das expressões digitadas na linha de comando interativa. Tecnicamente, essas também são instruções de expressão; elas servem como um atalho para digitar instruções de impressão.

A Tabela 8-5 lista algumas formas comuns de instrução de expressão do Python. As chamadas de funções e métodos são escritas com zero ou mais objetos argumento (na realidade, expressões que avaliam objetos) entre parênteses, após a função ou método.

Tabela 8-5 Instruções de expressão comuns do Python

Operação	Interpretação
`spam(eggs, ham)`	Chamadas de função
`spam.ham(eggs)`	Chamadas de método
`Spam`	Impressão interativa
`spam < ham and ham!= eggs`	Expressões compostas
`spam < ham < eggs`	Testes de intervalo

A última linha da tabela é uma forma especial: o Python nos permite enfileirar testes de comparação de grandeza para escrever comparações encadeadas, como os testes de intervalo. Por exemplo, a expressão `(A < B < C)` testa se `B` está entre `A` e `C`; isso é equivalente ao teste booleano `(A < B and B < C)`, mas é mais fácil de ler (e digitar). Normalmente, as expressões compostas não são escritas como instruções, mas fazer isso é sintaticamente

válido e pode até ser útil no prompt interativo, caso você não tenha certeza do resultado de uma expressão.

Tenha cuidado, porque, embora as expressões possam aparecer como instruções no Python, elas não podem ser usadas como expressões. Por exemplo, o Python não nos permite incorporar instruções de atribuição (`=`) em outras expressões. O fundamento lógico é que isso evita erros comuns de desenvolvimento; você não pode alterar uma variável acidentalmente, digitando `=`, quando na verdade queria usar o teste de igualdade `==`. Você vai ver como codificar em torno disso, quando conhecer o loop `while` do Pyhon, no Capítulo 10.

INSTRUÇÕES DE IMPRESSÃO

A instrução `print` simplesmente imprime objetos. Tecnicamente, ela grava a representação textual dos objetos no fluxo de saída padrão. O fluxo de saída padrão é o mesmo que a instrução `stdout` da linguagem C; normalmente, ele é mapeado na janela onde você iniciou seu programa em Python (a não ser que seja redirecionado para um arquivo no shell do seu sistema).

No Capítulo 7, vimos também métodos de arquivo que gravavam texto. A instrução `print` é semelhante, mas é mais focalizada: `print` grava objetos no fluxo de `stdout` (com alguma formatação padrão), mas os métodos de gravação de arquivo gravam strings nos arquivos. Como o fluxo de saída padrão está disponível no Python como o objeto `stdout`, no módulo interno `sys` (isto é, `sys.stdout`), é possível simular a instrução `print` com gravações de arquivo, mas `print` é mais fácil de usar.

A Tabela 8-6 lista as formas da instrução `print`. Já vimos a instrução `print` básica em ação. Por padrão, ela adiciona um espaço entre os itens separados por vírgulas e adiciona um avanço de linha no final da linha de saída corrente:

```
>>> x = 'a'
>>> y = 'b'
>>> print x, y
a b
```

Tabela 8-6 Formas da instrução print

Operação	Interpretação
`print spam, ham`	Imprime objetos em `sys.stdout`; adiciona um espaço entre eles.
`print spam, ham,`	Igual, mas não adiciona caractere de novo linha no final do texto.
`print >> myfile, spam, ham`	Envia texto para `myfile.write` e não para `sys.stdout.write`.

Essa formatação é apenas um padrão; você pode usá-la ou não. Para suprimir o avanço de linha (para que você possa adicionar mais texto na linha corrente posteriormente), finalize sua instrução `print` com uma vírgula, conforme mostrado na segunda linha da Tabela 8-6. Para suprimir o espaço entre os itens, você mesmo pode, em vez disso, construir uma string de saída, usando as ferramentas de concatenação e formatação de string abordadas no Capítulo 5, e imprimir a string toda de uma vez:

```
>>> print x + y
ab
>>> print '%s...s' % (x, y)
a...b
```

O programa "Hello World" do Python

Para imprimir uma mensagem "hello world", você usa apenas:

```
>>> print 'hello world'              # Imprime um objeto string.
hello world
```

Como os resultados de expressão são ecoados na linha de comando interativa, você, freqüentemente, não precisa usar uma instrução `print` lá; basta digitar as expressões que gostaria de imprimir e seus resultados serão ecoados:

```
>>> 'hello world'                    # Eco interativo
'hello world'
```

Na realidade, a instrução `print` é apenas um recurso ergonômico do Python – ela fornece uma interface amigável para o usuário para o objeto `sys.stdout`, com um pouco de formatação padrão. Você também pode codificar operações de impressão desta maneira:

```
>>> import sys                       # Imprimindo da maneira difícil
>>> sys.stdout.write('hello world\n')
hello world
```

Esse código chama explicitamente o método `write` de `sys.stdout` – um atributo configurado previamente, quando o Python inicia, para um objeto arquivo aberto, conectado no fluxo de saída. A instrução `print` oculta a maior parte desses detalhes. Ela fornece uma ferramenta simples para tarefas de impressão elementares.

Redirecionando o fluxo de saída

O equivalente de impressão de `sys.stdout` revela-se a base de uma técnica comum no Python. Em geral, `print` e `sys.stdout` se relacionam como segue:

```
print X
```

é equivalente à mais longa:

```
import sys
sys.stdout.write(str(X) + '\n')
```

que realiza uma conversão de string manual com `str`, adiciona um caractere de nova linha com + e chama o método `write` do fluxo de saída. A forma mais longa não é tão útil assim para impressão por si só. Entretanto, é interessante saber que é exatamente isso que as instruções `print` fazem, pois é possível reatribuir `sys.stdout` para algo diferente do fluxo de saída padrão. Em outras palavras, essa equivalência proporciona uma maneira de fazer suas instruções `print` enviarem texto para outros lugares. Por exemplo:

```
import sys
sys.stdout = open('log.txt', 'a')    # Redireciona impressões para arquivo
...
print x, y, x                        # Aparece em log.txt
```

Aqui, reconfiguramos `sys.stdout` para um objeto arquivo de saída aberto manualmente no modo *`append`. Após a reconfiguração, toda instrução `print`, em qualquer parte do programa, gravará seu texto no final do arquivo `log.txt`, em vez de gravar no fluxo de saída original. As

* N. de R.T.: Modo `append` insere dados sempre no final do arquivo.

instruções `print` gostam de ficar chamando o método `write` de `sys.stdout`, independente do que `sys.stdout` se refira.

Na verdade, conforme o quadro deste capítulo sobre `print` e `stdout` explicará, podemos até reconfigurar `sys.stdout` para objetos que não são arquivos, contanto que eles tenham o protocolo esperado (um método `write`); quando esses objetos são classes, o texto impresso pode ser direcionado e processado arbitrariamente.

Por que isto é relevante: print e stdout

É importante observar as equivalências entre a instrução `print` e gravar em `sys.stdout`. Isso torna possível reatribuir `sys.stdout` para um objeto definido pelo usuário que forneça os mesmos métodos que os arquivos (por exemplo, `write`). Como a instrução `print` apenas envia texto para o método `sys.stdout.write`, você pode capturar o texto impresso em seus programas, atribuindo `sys.stdout` a um objeto cujo método `write` processe o texto de maneiras arbitrárias.

Por exemplo, você pode enviar o texto impresso para uma janela de GUI, definindo um objeto com um método `write` que faça o direcionamento. Você verá um exemplo desse truque posteriormente no livro, mas, de forma abstrata, ele é semelhante ao seguinte:

```
class FileFaker:
    def write(self, string):
        # Faz algo com a string.

import sys
sys.stdout = FileFaker()
print someObjects           # Envia para o método write da classe
```

Isso funciona porque `print` é o que chamaremos de operação *polimórfica* na próxima parte deste livro e ela não se preocupa com o que é `sys.stdout`, mas apenas com o fato de que ele tem um método (isto é, interface) chamado `write`. Este redirecionamento para objetos é ainda mais simples:

```
myobj = FileFaker()
print >> myobj, someObjects              # Não reconfigura sys.stdout
```

A função interna `raw_input()` do Python lê o arquivo `sys.stdin`, de modo que você pode interceptar pedidos de leitura de maneira semelhante – usando classes que implementam métodos de leitura do tipo arquivo. Consulte o exemplo com `raw_input()` e de loop `while` no Capítulo 10, para ver mais fundamentos sobre isso.

Note que, como o texto de `print` vai para o fluxo `stdout`, essa é a maneira de imprimir código HTML em scripts CGI. Isso também significa que você pode redirecionar entrada e saída de script em Python na linha de comando do sistema operacional, como sempre:

```
python script.py < inputfile > outputfile
python script.py | filterProgram
```

A extensão print>>arquivo

Esse truque de redirecionamento de texto impresso, por meio da atribuição de `sys.stdout`, era tão comumente usado que a instrução `print` do Python tem uma extensão para torná-lo mais fácil. Um problema em potencial do código da última seção é que não há nenhuma ma-

neira direta de restaurar o fluxo de saída original, caso você precise voltar atrás após imprimir em um arquivo. Sempre podemos salvar e restaurar, conforme for necessário:*

```
import sys
temp = sys.stdout                        # Salva para restaurar.
sys.stdout = open('log.txt', 'a')        # Redireciona as impressões
                                         para o arquivo
...
print x, y, x                            # Imprime no arquivo.
...
sys.stdout = temp
print a, b, c                            # Imprime no stdout original.
```

Mas isso é complicado o bastante para que uma extensão `print` tenha sido acrescentada a fim de tornar o salvamento e a restauração desnecessários. Quando uma instrução `print` começa com `>>`, seguido de um objeto arquivo de saída (ou outro), essa única instrução `print` envia seu texto para o método `write` do objeto, mas não reconfigura `sys.stdout`. Como o redirecionamento é temporário, as instruções `print` normais continuam imprimindo no fluxo de saída original:

```
log = open('log.txt', 'a')
print >> log, x, y, z          # Imprime em um objeto do tipo arquivo.
print a, b, c                  # Imprime no stdout original.
```

A forma `>>` da instrução `print` é útil se você precisa imprimir em arquivos e no fluxo de saída padrão. Se você usar essa forma, certifique-se de fornecer a ela um objeto arquivo (ou um objeto que tenha o mesmo método `write` que um objeto arquivo) e não a string de nome de um arquivo.

* Também podemos usar o atributo relativamente novo `__stdout__` no módulo `sys`, que se refere ao valor original que `sys.stdout` tinha no momento da inicialização do programa. Contudo, ainda precisamos restaurar `sys.stdout` para `sys.__stdout__`, para voltarmos a esse valor de fluxo original. Consulte o módulo `sys` no manual da biblioteca para ver mais detalhes.

9

Testes if

Este capítulo apresenta a instrução if do Python – a principal instrução usada para selecionar ações alternativas com base em resultados de testes. Como esta é nossa primeira exposição às *instruções compostas* – instruções que incorporam outras instruções –, também exploraremos aqui os conceitos gerais existentes por trás do modelo da sintaxe de instrução do Python. Além disso, como a instrução if introduz a noção de testes, também usaremos este capítulo para estudarmos os conceitos de testes de verdade e expressões booleanas em geral.

INSTRUÇÕES if

Em termos simples, a instrução if do Python seleciona ações para executar. Ela é a principal ferramenta de linguagem e representa grande parte da *lógica* que um programa em Python possui. Ela também é nossa primeira instrução composta; assim como todas as instruções compostas do Python, if pode conter outras instruções, incluindo outras instruções if. Na verdade, o Python permite combinar instruções em um programa tanto seqüencialmente (para que sejam executadas uma após a outra) como arbitrariamente aninhadas (para que sejam executadas apenas sob certas condições).

Formato geral

A instrução if do Python é típica da maioria das linguagens procedurais. Ela assume a forma de um teste if, seguido de um ou mais testes elif opcionais (significando "else if"), e termina com um bloco else opcional. Cada teste e a cláusula else têm um bloco associado de instruções aninhadas, endentadas sob uma linha de cabeçalho. Quando a instrução é executada, o Python executa o bloco de código associado que possuir o primeiro teste avaliado como verdadeiro ou o bloco else, caso todos os testes sejam falsos. A forma geral de uma instrução if é a seguinte:

```
if <teste1>:                  # teste if
<instruções1>                 # Bloco associado
elif <teste2>:                # Instruções elif opcionais
<instruções2>
else                          # Instrução else opcional
< instruções3>
```

Exemplos

Vamos ver alguns exemplos simples da instrução `if` em funcionamento. Todas as partes são opcionais, exceto o teste `if` inicial e suas instruções associadas. No caso mais simples, as outras partes são omitidas:

```
>>> if 1:
...     print 'true'
...
true
```

Observe como o prompt muda para "...", para continuação de linhas na interface básica usada aqui (no IDLE, em vez disso, você simplesmente descerá para uma linha endentada – pressione Backspace para voltar para cima). Uma linha em branco finaliza e executa a instrução inteira. Lembre-se de que `1` é o valor verdadeiro booleano; portanto, o teste dessa instrução é sempre bem-sucedido; para tratar de um resultado falso, codifique a cláusula `else`:

```
>>> if not 1:
...     print 'true'
... else:
...     print 'false'
...
false
```

Agora, aqui está um exemplo do tipo mais complexo de instrução `if` – com todas as suas partes opcionais presentes:

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...     print "how's jessica?"
... elif x == 'bugs':
...     print "what's up doc?"
... else:
...     print 'Run away! Run away!'
...
Run away! Run away!
```

Essa instrução de várias linhas vai da linha `if` até o bloco `else`. Quando executada, o Python executa as instruções aninhadas sob o primeiro teste que seja verdadeiro ou a parte `else`, caso todos os testes sejam falsos (neste exemplo, eles são). Na prática, as partes `elif` e `else` podem ser omitidas e pode haver mais de uma instrução aninhada em cada seção. Além disso, as palavras `if`, `elif` e `else` são associadas pelo fato de serem alinhadas verticalmente, com a mesma endentação.

Desvio de vários caminhos

Se você já usou linguagens como C ou Pascal, pode estar interessado em saber que não existe nenhuma instrução "switch" ou "case" no Python, que seleciona uma ação com base no valor de uma variável. Em vez disso, o *desvio de vários caminhos* é codificado como uma série de testes `if`/`elif`, como foi feito no exemplo anterior, ou pela indexação de dicionários ou pesquisa em listas.

Como os dicionários e as listas podem ser construídos em tempo de execução, às vezes eles são mais flexíveis do que a lógica incorporada na instrução `if`:

```
>>> choice = 'ham'
>>> print {'spam': 1.25,          # Uma instrução 'switch' baseada em
                                    dicionário
```

```
...         'ham': 1.99,    # Usa has_key() ou get() por padrão
...         'eggs': 0.99,
...         'bacon': 1.10}[choice]
1.99
```

Embora normalmente leve alguns instantes para entender isso na primeira vez que você vê, esse dicionário é um desvio de vários caminhos – a indexação na chave `choice` desvia para um valor de um conjunto de valores, exatamente como uma instrução "switch" na linguagem C. Uma instrução `if` quase equivalente e mais longa do Python poderia ser a seguinte:

```
>>> if choice == 'spam':
...     print 1.25
... elif choice == 'ham':
...     print 1.99
... elif choice == 'eggs':
...     print 0.99
... elif choice == 'bacon':
...     print 1.10
... else:
...     print 'Bad choice'
...
1.99
```

Observe aqui a cláusula `else` na instrução `if` para tratar do caso *padrão*, quando nenhuma chave corresponde. Conforme vimos no Capítulo 6, os padrões de dicionários podem ser codificados com testes `has_key`, chamadas de método `get` ou captura de exceção. Todas as mesmas técnicas podem ser usadas aqui para codificar uma ação padrão em um desvio de vários caminhos baseado em dicionário. Aqui está o esquema do método `get` em funcionamento com padrões:

```
>>> branch = {'spam': 1.25,
...           'ham': 1.99,
...           'eggs': 0.99}
>>> print branch.get('spam', 'Bad choice')
1.25
>>> print branch.get('bacon', 'Bad choice')
Bad choice
```

Os dicionários são bons para associar valores a chaves, mas e quanto às ações mais complicadas que você pode codificar em instruções `if`? Na Parte IV, você vai aprender que os dicionários também podem conter *funções* para representar ações de desvio mais complexas, e implementam tabelas de salto gerais. Tais funções que aparecem como valores de dicionário, freqüentemente são escritas como `lambdas` e são chamadas pela adição de parênteses para ativar suas ações.

REGRAS DE SINTAXE DO PYTHON

Em geral, o Python tem uma sintaxe simples, baseada em instruções. Mas existem algumas propriedades que você precisa conhecer:

As instruções são executadas uma após a outra, até que você diga o contrário. Normalmente, o Python executa as instruções de um arquivo ou bloco aninhado, da primeira até a última, mas instruções como `if` (e, conforme você verá, os loops) fazem o interpretador saltar em seu código. Como o caminho do Python por um programa é chamado de fluxo de controle, comandos que o afetam (como a instrução `if`) são chamados de instruções de controle de fluxo.

Os limites de blocos e de instruções são detectados automaticamente. Não existem chaves ou delimitadores tipo "begin/end" em torno de blocos de código. Em vez disso, o Python usa a endentação de instruções sob um cabeçalho para agrupar as instruções em um bloco aninhado. Analogamente, as instruções do Python normalmente não terminam com um ponto-e-vírgula; em vez disso, o fim de uma linha marca o final das instruções escritas nessa linha.

Instruções compostas = cabeçalho. ":", instruções endentadas. No Python, todas as instruções compostas seguem o mesmo padrão: uma linha de cabeçalho terminada com dois-pontos, seguida de uma ou mais instruções aninhadas, normalmente endentadas sob o cabeçalho. As instruções endentadas são chamadas de *bloco* (ou, às vezes, de conjunto). Na instrução `if`, as cláusulas `elif` e `else` fazem parte dela, mas são linhas de cabeçalho com seus próprios blocos aninhados.

Normalmente, linhas em branco, espaços e comentários são ignorados. As linhas em branco são ignoradas em arquivos (mas não no prompt interativo). Os espaços dentro de instruções e expressões quase sempre são ignorados (exceto em literais de string e em endentação). Os comentários são sempre ignorados: eles começam com o caractere # (não dentro de uma literal de string) e se estendem até o fim da linha corrente.

As strings de documentação são ignoradas, mas são salvas e exibidas por ferramentas. O Python suporta uma forma de comentário adicional chamada string de documentação (abreviadamente, *docstring*), a qual, ao contrário dos comentários `#`, é mantida. As strings de documentação são simplesmente strings que aparecem no início de arquivos de programa e de algumas instruções, e são automaticamente associadas a objetos. Seu contéudo é ignorado pelo Python, mas elas são anexadas automaticamente aos objetos em tempo de execução, e podem ser exibidas com ferramentas de documentação. As `docstrings` fazem parte da documentação mais ampla do Python e serão abordadas no final da Parte III.

Conforme você viu, não existem declarações de tipo variável no Python. Esse fato sozinho contribui para uma sintaxe de linguagem muito mais simples do que você pode estar acostumado. Mas para a maioria dos usuários iniciantes, a falta de chaves e pontos-e-vírgulas para marcar blocos e instruções parece ser a característica sintática mais insólita do Python; portanto, vamos explorar o que isso significa com mais detalhes aqui.

Delimitadores de bloco

O Python detecta os limites automaticamente pela *endentação* da linha – o espaço vazio à esquerda de seu código. Todas as instruções endentadas com a mesma distância à direita pertencem ao mesmo bloco de código. Em outras palavras, as instruções dentro de um bloco são alinhadas verticalmente. O bloco termina em uma linha menos endentada ou no final do arquivo, e os blocos mais profundamente aninhados são simplesmente mais endentados à direita do que as instruções que estão no bloco que os engloba.

Por exemplo, a Figura 9-1 demonstra a estrutura de bloco do código a seguir:

```
x = 1
if x:
    y =2
    if y:
        print 'bloco2'
    print 'bloco1'
print 'bloco0'
```

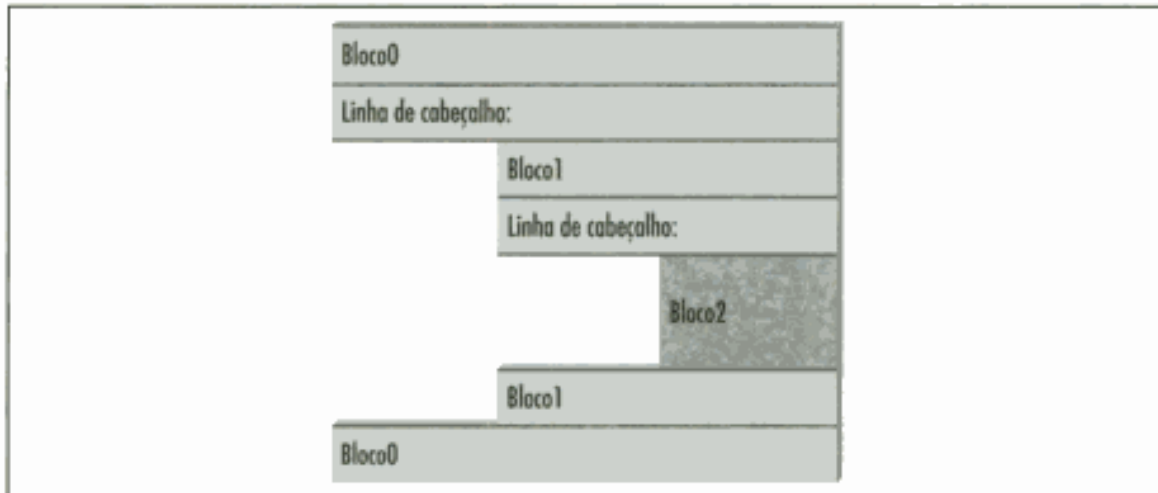



Figura 9-1 Blocos de código aninhados.

Esse código contém três blocos: o primeiro (o nível superior do arquivo) não é endentado, o segundo (dentro da instrução `if` mais externa) é endentado por quatro espaços e o terceiro (a instrução `print` sob a instrução `if` aninhada) é endentado por oito espaços.

Em geral, o código de nível superior (não aninhado) deve começar na linha 1. Os blocos aninhados podem começar em qualquer coluna; a endentação pode consistir em qualquer número de espaços e tabulações, contanto que seja a mesma para todas as instruções em determinado bloco. Isto é, o Python não se preocupa com a maneira como você endenta seu código; ele só se preocupa com o fato de que isso seja feito consistentemente. Tecnicamente, as tabulações valem por espaços suficientes para mover o número da coluna corrente para cima, em um múltiplo de 8, mas normalmente não é uma boa idéia misturar tabulações e espaços dentro de um bloco – use um ou outro.

A endentação à esquerda de seu código é o único lugar importante no Python onde o espaço em branco tem significado; na maioria dos outros contextos, o espaço pode ser escrito ou não. Entretanto, a endentação faz parte da sintaxe do Python e não é apenas uma sugestão estilística: todas as instruções dentro de determinado bloco devem ter a mesma endentação, senão o Python relatará um erro de sintaxe. Isso é assim de propósito – como você não precisa marcar explicitamente o início e o fim de um bloco de código aninhado, isso elimina parte da confusão sintática encontrada em outras linguagens.

Esse modelo de sintaxe também impõe a consistência da endentação, um componente fundamental da legibilidade em linguagens de programação estruturadas, como o Python. Às vezes, a sintaxe do Python é chamada de "o que você vê é o que obtém" das linguagens – a endentação de código informa aos leitores, inequivocamente, o que está associado com o que. A aparência consistente do Python torna o código mais fácil de manter.

Um código endentado consistentemente sempre satisfaz as regras do Python. Além disso, a maioria dos editores de textos (incluindo o IDLE) torna fácil seguir o modelo de endentação do Python, endentando o código automaticamente, enquanto você digita.

Delimitadores de instrução

Normalmente, as instruções terminam no fim da linha em que aparecem. Isso abrange a ampla maioria das instruções do Python que você escreverá. Contudo, quando as instruções são longas demais para caberem em uma única linha, algumas regras especiais podem ser usadas para fazer com que elas se estendam por várias linhas de continuação:

As instruções podem abranger várias linhas se você estiver continuando um par sintático aberto. Para instruções que são longas demais para caber em uma única linha, o Python permite que você continue a digitar a instrução na linha seguinte, caso esteja codificando algo incluído em pares `()`, `{}` ou `[]`. Por exemplo, expressões entre parênteses e literais de dicionário e lista podem abranger qualquer número de linhas. Sua instrução não termina até a linha na qual você digita a parte referente ao fechamento do par `()`, `}` ou `]`). A continuação de linhas pode começar em qualquer nível de endentação.

As instruções podem abranger várias linhas se terminarem com uma barra invertida. Este é um recurso um tanto desatualizado, mas se uma instrução precisa ocupar várias linhas, você também pode adicionar uma barra invertida (`\`) no final da linha anterior, para indicar que está continuando na linha seguinte. Mas como você também pode continuar adicionando parênteses em torno de construções longas, as barras invertidas quase nunca são necessárias.

Outras regras. Literais de string muito longas podem abranger linhas arbitrariamente. Na verdade, os blocos de string com aspas triplas, que conhecemos no Capítulo 5, são projetados para isso. Embora seja incomum, você também pode terminar instruções com um ponto-e-vírgula – às vezes isso é usado para espremer mais de uma instrução simples em uma única linha. Finalmente, comentários e linhas em branco podem aparecer em qualquer lugar.

Alguns casos especiais

Aqui temos um exemplo de como fica uma continuação de linha usando a regra dos pares abertos; podemos ocupar construções delimitadas com qualquer número de linhas:

```
L = ["Good",
     "Bad",
     "Ugly"]                    # Pares abertos podem abranger várias linhas.
```

Isso também funciona para tudo que estiver entre parênteses: expressões, argumentos de função, cabeçalhos de função (veja o Capítulo 12) etc. Se quiser usar barras invertidas para continuar, você pode, mas normalmente isso não é necessário:

```
if a == b and c == d and \
   d == e and f == g:
   print 'olde'                 # Barras invertidas permitem continuações.
```

Como qualquer expressão pode ser colocada entre parênteses, normalmente você pode simplesmente envolver algo entre parênteses sempre que precisar, para abranger várias linhas:

```
if (a == b and c == d and
    d == e and f == g):
    print 'new'                 # Mas parênteses normalmente também permitem.
```

Como um caso especial, o Python permite que você escreva mais de uma instrução não-composta (isto é, instruções sem outras aninhadas) na mesma linha, separadas por pontos-e-vírgulas. Alguns codificadores utilizam essa forma para economizar espaço em arquivos de programa, mas normalmente o código fica mais legível se você coloca uma instrução por linha na maior parte de seu trabalho:

```
x = 1; y = 2; print x                # Mais de uma instrução simples
```

Finalmente, o Python permite que você mova o corpo de uma instrução composta para a linha de cabeçalho, desde que o corpo seja apenas uma instrução simples (não-composta). Você vai ver isso usado com mais freqüência em instruções `if` simples, com um único teste e uma única ação:

```
if 1: print 'hello'        # Instrução simples na linha de cabeçalho
```

Você pode combinar alguns desses casos especiais para escrever código difícil de ler, mas não recomendamos isso. Como regra geral, tente manter cada instrução em sua própria linha e endente tudo, exceto os blocos mais simples. Com seis meses de trabalho, você ficará feliz de ter feito isso.

TESTES DE VERDADE

Apresentamos as noções de comparação, igualdade e valores de verdade no Capítulo 7. Como a instrução `if` é a primeira que realmente utiliza resultados de teste, expandiremos algumas dessas idéias aqui. Em particular, os operadores booleanos do Python são um pouco diferentes de seus correlatos em linguagens como C. No Python:

- Verdadeiro significa qualquer número diferente de zero ou objeto não-vazio.
- Falso significa não verdadeiro: um número zero, um objeto vazio ou `None`.
- As comparações e testes de igualdade são aplicados recursivamente nas estruturas de dados.
- As comparações e testes de igualdade retornam `1` ou `0` (verdadeiro ou falso).
- Os operadores booleanos `and` e `or` retornam um objeto operando verdadeiro ou falso.

Em resumo, os operadores booleanos são usados para combinar os resultados de outros testes. Existem três operações de expressão booleanas no Python:

`X and Y`
: É verdadeiro se `X` e `Y` são verdadeiros

`X or Y`
: É verdadeiro se `X` ou `Y` é verdadeiro

`not X`
: É verdadeiro se `X` é falso (a expressão retorna `1` ou `0`)

Aqui, `X` e `Y` podem ser qualquer valor de verdade ou uma expressão que retorna um valor de verdade (por exemplo, um teste de igualdade, comparação de intervalo etc.). Os operadores booleanos são digitados como palavras no Python (em vez de `&&`, `||` e `!` da linguagem C). No Python, os operadores booleanos `and` e `or` retornam um *objeto* verdadeiro ou falso. Vamos ver alguns exemplos para saber como isso funciona:

```
>>> 2 < 3, 3 < 2                    # Menor que: retorna 1 ou 0
(1, 0)
```

Comparações de grandeza como essas retornam um valor inteiro `1` ou `0` como resultado do valor de verdade. Mas os operadores `and` e `or`, em vez disso, sempre retornam um objeto. Para testes do operador `or`, o Python avalia os objetos operando da esquerda para a direita e retorna o primeiro que for verdadeiro. Além disso, o Python pára no primeiro operando verdadeiro que encontra; normalmente, isso é chamado de *avaliação de curto-circuito*, pois a determinação de um resultado causa um curto-circuito (termina) no restante da expressão:

```
>>> 2 or 3, 3 or 2          # Retorna o operando da esquerda se for
                              verdadeiro.
(2, 3)                      # Senão, retorna o operando da direita
                              (verdadeiro ou falso).
>>> [] or 3
3
>>> [] or {}
{}
```

Na primeira linha anterior, os dois operandos são verdadeiros (2,3); portanto, o Python sempre pára e retorna o que está à esquerda. Nos outros dois testes, o operando da esquerda é falso; portanto, o Python simplesmente avalia e retorna o objeto da direita (que terá um valor verdadeiro ou falso, se for testado). Além disso, as operações de and param assim que o resultado for conhecido; neste caso, o Python avalia os operandos da esquerda para a direita e pára no primeiro objeto falso:

```
>>> 2 and 3, 3 and 2        # Retorna o operando da esquerda se for falso.
(3, 2)                      # Senão, retorna o operando da direita
                              (verdadeiro ou falso).
>>> [] and {}
[]
>>> 3 and []
[]
```

Aqui, os dois operandos são verdadeiros na primeira linha; portanto, o Python avalia os dois lados e retorna o objeto da direita. No segundo teste, o operando da esquerda é falso ([]); portanto, o Python pára e o retorna como resultado do teste. No último teste, o lado esquerdo é verdadeiro (3); portanto, o Python avalia e retorna o objeto da direita (que por acaso é falso []).

O resultado final de tudo isso é o mesmo em C e na maioria das outras linguagens – você recebe um valor que é logicamente verdadeiro ou falso, se testado em uma instrução if ou while. Entretanto, no Python, os valores booleanos retornam o objeto da esquerda ou da direita e não um flag inteiro.

Uma nota final: conforme descrito no Capítulo 7, o Python 2.3 inclui um novo tipo booleano, chamado bool, que internamente é uma subclasse do tipo inteiro int, com valores True e False. Esses dois casos são, na verdade, apenas versões personalizadas dos valores inteiros 1 e 0, os quais produzem as palavras True e False quando impressos ou convertidos em strings de alguma outra forma. A única vez em que você geralmente notará essa mudança será quando ver saídas booleanas impressas como True e False. Mais informações sobre subclasses de tipo aparecem no Capítulo 23.

Por que isto é relevante: valores booleanos

Uma maneira comum de usar o comportamento exclusivo dos operadores booleanos do Python é na seleção de um conjunto de objetos com um operador `or`. Uma instrução:

```
X = A or B or C or None
```

configura `X` com o primeiro objeto não-vazio (ou seja, verdadeiro) entre `A`, `B` e `C`, ou `None`, se todos forem vazios. Isso se mostra um paradigma de desenvolvimento muito comum no Python: para selecionar um objeto não-vazio em um conjunto de tamanho fixo, basta enfileirá-los em uma expressão `or`.

Também é importante entender a avaliação de curto-circuito, pois as expressões à direita de um operador booleano poderiam chamar funções que fazem muito trabalho ou têm efeitos colaterais que não aconteceriam se a regra do curto-circuito surtisse efeito:

```
if f1() or f2(): ...
```

Aqui, se `f1` retornar um valor verdadeiro (ou não-vazio), o Python nunca executará `f2`. Para garantir que as duas funções sejam executadas, chame-as antes do operador `or`:

```
tmp1, tmp2 = f1(), f2()
if tmp1 or tmp2: ...
```

Você verá outra aplicação desse comportamento no Capítulo 14: por causa da maneira como os valores booleanos funcionam, a expressão (`(A and B) or C`) pode ser usada para simular uma instrução `if/else` – ou quase. Observe também que, como todos os objetos são inerentemente verdadeiros ou falsos, no Python é comum e mais fácil testar um objeto diretamente (`if X:`), em vez de compará-lo com um valor vazio (`if X!= '':`). Para uma string, os dois testes são equivalentes.

10

Loops while e for

Neste capítulo, conheceremos as duas principais construções de *loops* do Python – instruções que reproduzem uma ação repetidamente. A primeira delas, o loop `while`, fornece uma maneira de desenvolver loops gerais. A segunda, o loop `for`, é projetada para percorrer os itens de um objeto de seqüência e executar um bloco de código para cada item.

Existem outros tipos de operações de loop no Python, mas as duas instruções abordadas aqui representam a principal sintaxe fornecida para escrever ações repetidas. Também estudaremos aqui algumas instruções incomuns, como `break` e `continue`, pois elas são usadas dentro de loops.

LOOPS WHILE

A instrução `while` do Python é sua construção de iteração mais geral. Em termos simples, ela executa repetidamente um bloco de instruções endentadas, contanto que um teste no início continue avaliando um valor verdadeiro. Quando o teste se torna falso, o controle continua após todas as instruções presentes no bloco `while`; o miolo nunca é executado se o teste é falso desde o início.

A instrução `while` é uma das duas instruções de loop (junto com `for`). Ela é chamada de loop porque o controle continua voltando ao início da instrução, até que o teste se torne falso. O resultado é que o miolo do loop é executado repetidamente, enquanto o teste que está no início for verdadeiro. Além das instruções, o Python também fornece várias ferramentas que fazem loop (iteram) implicitamente: as funções `map`, `reduce` e `filter`, o teste de participação como membro `in`, compreensões de lista e muito mais. Vamos explorar a maior parte delas no Capítulo 14.

Formato geral

Em sua forma mais complexa, a instrução `while` consiste em uma linha de cabeçalho com uma expressão de teste, um miolo com uma ou mais instruções endentadas e uma parte `else` opcional, que é executada se o controle sai do loop sem passar por uma instrução `break`.

O Python continua avaliando o teste do início e executando as instruções aninhadas na parte `while`, até que o teste retorne um valor falso:

```
while <teste>:              # Faz um loop em teste
    <instruções1>           # Miolo do loop
else:                       # else opcional
    <instruções2>           # Executadas se não saiu do loop com break
```

Exemplos

Para ilustrar, aqui estão alguns loops `while` simples em ação. O primeiro apenas imprime uma mensagem para sempre, aninhando uma instrução `print` em um loop `while`. Lembre-se de que um valor inteiro `1` significa verdadeiro; como o teste é sempre verdadeiro, o Python continua executando o miolo para sempre ou até que você interrompa sua execução. Normalmente, esse tipo de comportamento é chamado de *loop infinito*:

```
>>> while 1:
...     print 'Type Ctrl-C to stop me!'
```

O próximo exemplo fica fracionando o primeiro caractere de uma string, até que a string esteja vazia e, portanto, falsa. É comum testar um objeto diretamente dessa forma, em vez de usar o equivalente mais longo: `while x != ' ':`. Posteriormente neste capítulo, veremos outras maneiras de percorrer mais diretamente os itens de uma string com um loop `for`.

```
>>> x = 'spam'
>>> while x:
...     print x,
...     x = x[1:]      # Retira o primeiro caractere de x.
...
spam pam am m
```

O código a seguir conta do valor de `a` até (mas não incluindo) `b`. Posteriormente, veremos uma maneira mais fácil de fazer isso com as instruções `for` e `range` do Python.

```
>>> a=0; b=10
>>> while a < b:       # Uma maneira de codificar loops contadores
...     print a,
...     a += 1         # Ou a = a+1
...
0 1 2 3 4 5 6 7 8 9
```

BREAK, CONTINUE, PASS E A CLÁUSULA ELSE

Agora que já vimos nosso primeiro loop em Python, devemos apresentar duas instruções simples que só tem significado quando aninhadas dentro de loops – as instruções `break` e `continue`. Estudaremos também a cláusula de loop `else`, pois ela é entrelaçada com `break`, e a instrução de lugar reservado vazio, `pass`. No Python:

`break`
Sai do loop mais próximo que a envolve (após a instrução de loop inteira)

`continue`
Pula para o início do loop mais próximo que a envolve (para a linha de cabeçalho do loop)

`pass`
Não faz absolutamente nada; trata-se de um lugar reservado de instrução, vazio

Bloco `else` *do loop*
É executado se, e somente se, saímos do loop normalmente – sem atingir uma instrução `break`

Formato de loop geral

Levando em conta as instruções `break` e `continue`, o formato geral do loop `while` é o seguinte:

```
while <teste1>:
    <instruções1>
    if <teste2>: break          # Sai do loop agora, pula a cláusula else.
    if <teste3>: continue       # Vai para o início do loop agora.
else:
    <instruções2>               # Se não atingimos uma instrução 'break'
```

As instruções `break` e `continue` podem aparecer em qualquer lugar dentro do miolo do loop `while` (e `for`), mas normalmente são escritas de forma mais aninhada em um teste `if`, para entrarem em ação em resposta a algum tipo de condição.

Exemplos

Vamos ver alguns exemplos simples para saber como essas instruções aparecem juntas na prática. A instrução `pass` é usada quando a sintaxe exige uma instrução, mas você não tem nada de útil a fazer. Ela é usada freqüentemente para escrever um miolo vazio de uma instrução composta. Por exemplo, se você quiser escrever um loop infinito que não faça nada sempre que for percorrido, utilize a instrução `pass`:

```
while 1: pass   # Digite Ctrl-C para interromper!
```

Como o miolo é apenas uma instrução vazia, o Python fica preso nesse loop.* De forma aproximada, a instrução `pass` está para as instruções assim como `None` está para os objetos – um nada explícito. Note que o miolo do loop `while` está na mesma linha do cabeçalho, após os dois-pontos. Assim como na instrução `if`, isso só funciona se o miolo não for uma instrução composta.

A instrução `continue` é um salto imediato para o início de um loop. Às vezes, ela permite que você evite o aninhamento de instruções. O próximo exemplo usa `continue` para pular números ímpares. Esse código imprime todos os números pares menores do que `10` e maiores ou iguais a 0. Lembre-se de que 0 significa falso e `%` é o operador de resto de divisão; portanto, esse loop conta regressivamente até zero, pulando os números que não são múltiplos de dois (ele imprime 8 6 4 2 0):

```
x = 10
while x:
    x = x-1                     # Ou x-= 1
    if x % 2 != 0: continue     # Ímpar?--pula a impressão
    print x,
```

Como a instrução `continue` pula para o início do loop, você não precisa aninhar a instrução `print` dentro de um teste `if`; a instrução `print` só é alcançada se a instrução `continue` não é executada. Se isso parece familiar com uma instrução "goto" de outras linguagens, está certo. O Python não tem nenhuma instrução `goto`, mas como a instrução `continue` permite que você salte em um programa, muitos dos alertas sobre a legibilidade e a manutenção, que você pode

* Esse código não faz nada, para sempre. Provavelmente, esse não é o programa em Python mais útil já escrito, a não ser que você queira testar um medidor de CPU ou aquecer seu computador laptop em um dia frio de inverno. Francamente, contudo, não conseguimos imaginar um exemplo melhor com a instrução `pass`. Veremos outros lugares onde ela faz sentido, posteriormente no livro (por exemplo, no Capítulo 22, para definir classes vazias que implementam objetos que se comportam como "estruturas" e "registros" em outras linguagens).

ter ouvido sobre a instrução `goto`, se aplicam. Provavelmente, a instrução `continue` deve ser pouco usada, especialmente quando você está começando a aprender sobre Python. O exemplo anterior poderia ser mais claro se a instrução `print` estivesse aninhada sob a instrução `if`, por exemplo:

```
x = 10
while x:
    x = x-1
    if x % 2 == 0:              # Par?--imprime
        print x,
```

A instrução `break` é uma saída imediata do loop. Como o código que está depois dela nunca é atingido, às vezes a instrução `break` também pode evitar o aninhamento. Por exemplo, aqui está um loop interativo simples, o qual insere dados com `raw_input` e termina quando o usuário digita "stop" para o nome solicitado:

```
>>> while 1:
...     name = raw_input('Enter name:')
...     if name == 'stop': break
...     age  = raw_input('Enter age: ')
...     print 'Hello', name, '=>', int(age) ** 2
...
Enter name:mel
Enter age: 40
Hello mel => 1600
Enter name:bob
Enter age: 30
Hello bob => 900
Enter name:stop
```

Observe como esse código converte a entrada de `age` em um inteiro, antes de elevá-la à segunda potência, com `int`; `raw_input` retorna a entrada do usuário como uma string. No Capítulo 22, você verá que ela também pode lançar uma exceção no final do arquivo (por exemplo, se o usuário digitar `Ctrl-Z` ou `Ctrl-D`). Se isso importa, englobe `raw_input` em instruções `try`.

Quando combinada com a cláusula `else`, a instrução `break` freqüentemente pode eliminar os flags de status de pesquisa usados em outras linguagens. Por exemplo, o trecho de código a seguir determina se um número inteiro positivo `y` é primo, procurando fatores maiores do que `1`:

```
x = y / 2                       # Para algum y > 1
while x > 1:
    if y % x == 0:              # Resto
        print y, 'has factor', x
        break                   # Pula a cláusula else
    x = x-1
else:                           # Saída normal
    print y, 'is prime'
```

Em vez de configurar um flag para ser testado quando o loop termina, insira uma instrução `break` onde um fator é encontrado. Desse modo, a cláusula `else` pode supor que será executada somente se nenhum fator for encontrado. Se você não atingir a instrução `break`, o número é primo.*

* Mais ou menos. Números menores do que `2` não são considerados primos pela definição matemática estrita. Sendo realmente rigorosos, esse código também falha para números negativos e em ponto flutuante, além de ser arruinado pela alteração da futura divisão "real" `/`, mencionada no Capítulo 4. Se você quiser experimentar esse código, veja o exercício no final da Parte 4, que o engloba em uma função.

A cláusula `else` também será executada se o miolo do loop nunca for executado, pois você também não executa uma instrução `break` nesse caso. Em um loop `while`, isso acontece se o teste no cabeçalho é falso logo no início. No exemplo anterior, você ainda recebe a mensagem `"is prime"` (é primo) se `x` for inicialmente menor ou igual a `1` (por exemplo, se `y` for `2`).

Mais informações sobre a cláusula else

Como a cláusula `else` é única no Python, à primeira vista ela tende a confundir alguns iniciantes. Em termos mais gerais, a cláusula `else` fornece uma sintaxe explícita para um cenário de desenvolvimento comum – ela é uma estrutura de desenvolvimento que permite a você capturar a "outra" maneira de sair de um loop, sem configurar nem verificar flags ou condições.

Suponha, por exemplo, que você esteja escrevendo um loop para procurar um valor em uma lista e precise saber se o valor foi encontrado após sair do loop. Você poderia escrever essa tarefa da seguinte maneira:

```
found = 0
while x and not found:
    if match(x[0]):             # O valor está no início?
        print 'Ni'
        found = 1
    else:
        x = x[1:]               # Fraciona o início e repete.
if not found:
    print 'not found'
```

Aqui, inicializamos, configuramos e, posteriormente, testamos um flag, para saber se a pesquisa foi bem-sucedida. Esse é um código Python válido e funciona, mas é exatamente o tipo de estrutura para a qual a cláusula foi feita para manipular. Aqui está um código equivalente com a cláusula `else`:

```
while x:                        # Sai quando x está vazio.
    if match(x[0]):
        print 'Ni'
        break                   # Sai, vai para outro lugar.
    x = x[1:]
else:
    print 'Not found            # Aqui, somente se esgotou x.
```

Aqui, o flag desapareceu e substituímos o teste `if` no fim do loop por uma instrução `else` (alinhada verticalmente com a palavra `while`, pela endentação). Como a instrução `break`, dentro da parte principal da instrução `while`, sai do loop e rodeia a instrução `else`, isso serve como uma maneira mais estruturada de capturar o caso de falha da pesquisa.

Alguns leitores podem notar que a cláusula `else` do exemplo anterior poderia ser substituída por um teste de `x` vazio após o loop (por exemplo, `if not x:`). Embora isso seja verdade nesse exemplo, a cláusula `else` fornece uma sintaxe explícita para esse padrão de codificação (é mais obviamente uma cláusula de falha de pesquisa aqui) e, em alguns casos, esse teste de vazio explícito pode não se aplicar. Além disso, a cláusula `else` se torna ainda mais útil quando usada em conjunto com o loop `for`, pois a iteração da seqüência não está sob seu controle.

LOOPS for

O loop `for` é um iterador de seqüência genérico no Python: ele pode percorrer os itens de qualquer objeto seqüência ordenada. O loop `for` funciona em strings, listas, tuplas e em novos objetos que criaremos posteriormente com classes.

Formato geral

O loop `for` do Python começa com uma linha de cabeçalho que especifica um destino (ou destinos) de atribuição, junto com um objeto que você queira percorrer. O cabeçalho é seguido por um bloco de instruções endentadas que você queira repetir:

```
for <destino> in <objeto>:      # Atribui itens do objeto ao destino.
    <instruções>                # Miolo do loop repetido: usa o destino
else:
    <instruções>                # Se não atingirmos uma instrução 'break'
```

Quando o Python executa um loop `for`, ele atribui os itens do objeto seqüência ao *destino*, um por um, e executa o miolo do loop para cada um. Normalmente, o miolo do loop usa o

Por que isto é relevante: simulando loops while da linguagem C

A seção sobre instruções de expressão afirmou que o Python não permite que instruções, como as atribuições, apareçam em lugares onde ela espera uma expressão. Isso significa que um padrão de desenvolvimento comum da linguagem C não funcionaria no Python:

```
while ((x = next()) != NULL) {...processa x...}
```

As atribuições da linguagem C retornam o valor atribuído; as atribuições do Python são apenas instruções e não expressões. Isso elimina uma notória classe de erros da linguagem C (você não pode digitar `=` acidentalmente no Python, quando quiser escrever `==`). Mas se você precisa de um comportamento semelhante, existem pelo menos três maneiras de obter o mesmo efeito nos loops `while` do Python, sem incoporar atribuições em testes de loop. Você pode mover a atribuição para o miolo do loop com uma instrução `break`:

```
while 1:
    x = next()
    if not x: break
    ...processa x...
```

mover a atribuição para o loop com testes:

```
x = 1
while x:
    x = next()
    if x:
        ...processa x...
```

ou mover a primeira atribuição para fora do loop:

```
x = next()
while x:
    ...processa x...
    x = next()
```

Desses três padrões de desenvolvimento, o primeiro pode ser considerado, por alguns, como o menos estruturado, mas também parece ser o mais simples e mais comumente utilizado. Um loop `for` simples do Python também pode substituir alguns loops da linguagem C.

destino da atribuição para se referir ao item corrente na seqüência, como se fosse um cursor percorrendo uma seqüência.

O nome usado como destino da atribuição em uma linha de cabeçalho de loop `for` normalmente é uma variável (possivelmente nova), no escopo onde o loop `for` é escrito. Não há nada de muito especial nisso; ela pode até ser alterada dentro do miolo do loop, mas será configurada automaticamente como o próximo item da seqüência, quando o controle retornar novamente para o início do loop. Após o loop, essa variável normalmente ainda se refere ao último item visitado, que é o último item da seqüência, a menos que se saia do loop com uma instrução `break`.

O loop `for` também aceita um bloco `else` opcional, que funciona exatamente como acontece nos loops `while`; ele é executado se o loop termina sem encontrar uma instrução `break` (isto é, se todos os itens da seqüência foram visitados). Na verdade, as instruções `break` e `continue`, apresentadas anteriormente, funcionam da mesma forma no loop `for` e no loop `while`. O formato completo do loop `for` pode ser descrito da seguinte maneira:

```
for <destino> in <objeto>:      # Atribui itens do objeto ao destino.
    <instruções>
    if <teste>: break           # Sai do loop agora, pula a cláusula else.
    if <teste>: continue        # Vai para o início do loop agora.
else:
    <instruções>                # Se não atingimos uma instrução 'break'
```

Exemplos

Vamos digitar alguns loops `for` interativamente. No primeiro exemplo, o nome `x` é atribuído a cada um dos três itens da lista por sua vez, da esquerda para a direita, e a instrução `print` é executada para cada um. Dentro da instrução `print` (o miolo do loop), o nome `x` refere-se ao item corrente na lista:

```
>>> for x in ["spam", "eggs", "ham"]:
...     print x,
...
spam eggs ham
```

Os próximos dois exemplos calculam a soma e o produto de todos os itens de uma lista. No próximo capítulo, veremos as funções internas que aplicam operações como + e * a itens de uma lista automaticamente, mas normalmente é fácil usar apenas um loop `for`:

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in[1, 2, 3, 4]:prod *= item
...
>>> prod
24
```

Os loops `for` também funcionam em strings e tuplas – qualquer seqüência funciona em um loop `for`:

```
>>> S, T = "lumberjack", ("and", "I'm", "okay")

>>> for x in S: print x,
...
l u m b e r j a c k

>>> for x in T: print x,
...
and I'm okay
```

Se você estiver fazendo uma iteração por uma seqüência de tuplas, o destino do loop pode ser, na verdade, uma *tupla* de destinos. Esse é apenas outro caso de atribuição de desempacotamento de tupla em funcionamento; lembre-se de que o loop `for` atribui itens da seqüência ao destino e a atribuição funciona de maneira igual em todos os lugares:

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:              # Atribuição de tupla em funcionamento
...     print a, b
...
1 2
3 4
5 6
```

Aqui, na primeira passagem pelo loop, é como escrever `(a,b) = (1,2)`. Na segunda, `(a,b)` é atribuído a `(3,4)` e assim por diante. Esse não é um caso especial; qualquer destino de atribuição funciona sintaticamente após a palavra `for`.

Agora, vamos ver algo um pouco mais sofisticado. O próximo exemplo ilustra a cláusula `else` em um loop `for` e o aninhamento de instruções. Dada uma lista de objetos (itens) e uma lista de chaves (tests), este código pesquisa cada chave na lista de objetos e relata o resultado da pesquisa:

```
>>> items = ["aaa", 111, (4, 5), 2.01     # Um conjunto de objetos
>>> tests = [(4, 5), 3.14]                # Chaves para pesquisar
>>>
>>> for key in tests:                     # Para todas as chaves
...     for item in item:                 # Para todos os itens
...         if item == key:               # Verifica a correspondência.
...             print key, "was found"
...             break
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```

Como a instrução `if` aninhada executa uma instrução `break` quando uma correspondência é encontrada, a cláusula `else` do loop pode supor que a pesquisa falhou. Observe o aninhamento aqui: quando esse código é executado, existem dois loops ocorrendo ao mesmo tempo. O loop externo percorre a lista de chaves e o loop interno percorre a lista de itens para cada chave. O aninhamento da cláusula `else` é fundamental; ela está endentada no mesmo nível da linha de cabeçalho do loop `for` interno; portanto, é associada ao loop interno (e não ao `if` ou ao `for` externo).

A propósito, esse exemplo será mais fácil de escrever se empregarmos o operador `in` para testar a participação como membro. Como o operador `in` percorre uma lista implicitamente, em busca de uma correspondência, ele substitui o loop interno:

```
>>> for key in tests:          # Para todas as chaves
...     if key in items:       # Deixa a Python procurar uma correspondência.
...         print key, "was found"
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```

Em geral, é uma boa idéia deixar o Python fazer o máximo de trabalho possível, por questões de brevidade e desempenho. O próximo exemplo executa uma tarefa típica de estrutura de dados com um loop `for` – coletar itens comuns em duas seqüências (strings). É mais ou menos uma rotina de interseção de conjuntos simples; após a execução do loop, `res` refere-se a uma lista que contém todos os itens encontrados em `seq1` e em `seq2`:

```
>>> seq1 = "spam"
>>> seq2 = "scam"
>>> res = []                   # Começa vazio.
>>> for x in seq1:             # Percorre a primeira seqüência.
...     if x in seq2:          # Item comum?
...         res.append(x)      # Adiciona no final do resultado.
...
>>> res
['s', 'a', 'm']
```

Infelizmente, esse código está equipado para funcionar apenas com duas variáveis específicas: `seq1` e `seq2`. Seria ótimo se esse loop pudesse ser generalizado de alguma forma, em uma ferramenta que você pudesse usar mais de uma vez. Conforme você verá, essa idéia simples nos leva às funções, o assunto da Parte 4.

VARIAÇÕES DE LOOP

O loop `for` subordina a maioria dos loops estilo contador. Geralmente ele é mais simples de escrever e mais rápido para executar do que um loop `while`, de modo que é a primeira ferramenta que você deve procurar quando precisar percorrer uma seqüência. Mas também existem situações em que você precisará iterar de uma maneira mais especializada. Por exemplo, e se você precisar visitar todo segundo e terceiro item de uma lista ou alterar a lista no processo? E quanto a percorrer mais de uma seqüência em paralelo no mesmo loop `for`?

Você sempre pode escrever tais iterações exclusivas com um loop while e indexação manual, mas o Python fornece duas funções internas que permitem especializar a iteração em um loop `for`:

- A função interna `range` retorna uma lista de inteiros sucessivamente mais altos, que podem ser usados como índices em um loop `for`.*
- A função interna `zip` retorna uma lista de tuplas de itens paralelos, que podem ser usadas para percorrer várias seqüências em um loop `for`.

Vamos ver cada uma dessas funções internas por sua vez.

* O Python também fornece uma função interna chamada `xrange` que gera um índice por vez, em vez de armazená-los simultaneamente em uma lista, como faz a função `range`. Não há nenhuma vantagem na velocidade de `xrange`, mas ela é útil como uma otimização de espaço, caso você precise gerar um número muito grande de valores.

Loops contadores: range

A função `range` é independente dos loops `for`. Embora seja mais freqüentemente usada para gerar índices em um loop `for`, você pode usá-la sempre que precise de uma lista de inteiros:

```
>>> range(5), range(2, 5), range(0, 10, 2)
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
```

Por que isto é relevante: varredores de arquivo

Em geral, os loops são úteis em qualquer lugar onde você precise repetir ou processar algo mais de uma vez. Como os arquivos contêm vários caracteres e linhas, eles são um dos usos mais típicos para os loops. Para carregar todo o conteúdo de um arquivo em uma string, de uma só vez, você simplesmente chama a instrução `read`:

```
file = open('test.txt', 'r')
print file.read()
```

Mas para carregar um arquivo por partes, é comum escrever um loop `while` com instruções `break` no final do arquivo ou um loop `for`. Para ler por caracteres:

```
file = open('test.txt')
while 1:
    char = file.read(1)          # Lê por caractere.
    if not char: break
    print char,

for char in open('test.txt').read():
    print char
```

O loop `for` aqui também processa cada caractere, mas carrega o arquivo na memória, todo de uma vez. Para ler por linhas ou blocos com um loop `while`:

```
file = open('test.txt')
while 1:
    line = file.readline()       # Lê linha por linha.
    if not line: break
    print line,

file = open('test.txt', 'rb')
while 1:
    chunk = file.read(10)        # Lê trechos de byte.
    if not chunk: break
    print chunk,
```

Contudo, para ler arquivos de texto linha por linha, o loop `for` tende a ser mais fácil de escrever e mais rápido para executar:

```
for line in open('test.txt').readlines(): print line
for line in open('test.txt').xreadlines():print line
for line in open('test.txt'): print line
```

`readlines` carrega um arquivo todo de uma vez em uma lista de string de linha; `xreadlines`, em vez disso, carrega as linhas de acordo com a demanda, para não encher a memória no caso de arquivos grandes. O último exemplo aqui conta com os novos *iteradores* de arquivo para obter o equivalente de `xreadlines` (os iteradores serão abordados no Capítulo 14). A partir do Python 2.2, o nome `open`, em todos os exemplos anteriores também, pode ser substituído pelo nome `file`. Consulte o manual da biblioteca para ver mais informações sobre as chamadas usadas aqui. Como regra geral, quanto maior o tamanho dos dados que você lê em cada passo, mais rapidamente seu programa é executado.

Com um argumento, a função range gera uma lista com inteiros de zero até (mas não incluindo) o valor do argumento. Se você passar dois argumentos, o primeiro será considerado como o limite inferior. Um terceiro argumento opcional pode fornecer um *passo*; se for usado, o Python soma o passo a cada inteiro sucessivo no resultado (o padrão dos passos é um). Os intervalos também podem ser negativos, em ordem descendente, caso você queira:

```
>>> range(-5, 5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4,]

>>> range(5, -5, -1)
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

Embora todos esses resultados de intervalo possam ser úteis por si só, eles tendem a ser mais úteis dentro de loops for. Por exemplo, eles fornecem uma maneira simples de repetir uma ação por um número específico de vezes. Para imprimir três linhas, por exemplo, use um intervalo para gerar o número de inteiros apropriado:

```
>>> for i in range(3):
...     print i, 'Pythons'
...
0 Pythons
1 Pythons
2 Pythons
```

A função range também é comumente usada para fazer iteração em uma seqüência, indiretamente. A maneira mais fácil e mais rápida de percorrer uma seqüência exaustivamente sempre é com um loop for simples. O Python trata da maior parte dos detalhes para você:

```
>>> X = 'spam'
>>> for item in X: print item,          # Iteração simples
...
s p a m
```

Observe a vírgula no final da instrução print aqui, para suprimir o avanço de linha padrão (cada impressão continua adicionando na linha de saída corrente). Internamente, o loop for trata dos detalhes da iteração automaticamente. Se você precisar realmente assumir explicitamente o controle da lógica de indexação, pode fazer isso com um loop while:

```
>>> i = 0
>>> while i < len(X):                   # iteração do loop while
...     print X[i],; i += 1
...
s p a m
```

Você também pode fazer indexação manual com um loop for, se usar a função range para gerar uma lista de índices para fazer a iteração:

```
>>> X
'spam'
>>> len(X)                              # Comprimento da string
4
>>> range(len(X))                       # Todos os deslocamentos válidos em X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): printX[i],  # Indexação de loop for manual
...
s p a m
```

O exemplo aqui está percorrendo uma lista de *deslocamentos* em X e não os *itens* reais de X. Precisamos indexar de volta em X, dentro do loop, para buscar cada item.

Varreduras não exaustivas: range

O último exemplo da seção anterior funciona, mas provavelmente é executado de forma mais lenta do que deveria. A não ser que você tenha um requisito de indexação especial, é sempre melhor usar a forma de loop `for` simples no Python – use o loop `for`, em vez do loop `while` quando possível, e não conte com chamadas de `range` em loops `for`, exceto como,último recurso.

Entretanto, o mesmo padrão de desenvolvimento usado naquele exemplo anterior também nos permite fazer tipos de varreduras mais especializadas:

```
>>> S = 'abcdefghijk'
>>> range(0, len(S), 2)
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print S[i],
...
a c e g i k
```

Aqui, visitamos cada *segundo* item na string `S`, percorrendo a lista gerada por `range`. Para visitar cada terceiro item, mude o terceiro argumento de `range` para `3` e assim por diante. Na verdade, a função `range` usada dessa maneira permite que você pule itens em loops, enquanto ainda mantém a simplicidade do loop `for`. Consulte também o novo terceiro limite de fracionamento opcional do Python 2.3, na seção "Indexação e fracionamento" do Capítulo 5. Na versão 2.3, um efeito semelhante pode ser obtido com:

```
for x in S[::2]: print x
```

Alterando listas: range

Outro lugar comum em que você pode usar a função `range` e o loop `for` combinados é em loops que alteram uma lista enquanto ela está sendo percorrida. O exemplo a seguir precisa de um índice para poder atribuir um valor atualizado a cada posição, à medida que avançamos:

```
>>> L = [1, 2, 3, 4, 5]
>>>
>>> for i in range(len(L)):          # Soma um a cada item em L
...     L[i] += 1                    # Ou L[i] = L[i] + 1
...
>>> L
[2, 3, 4, 5, 6]
```

Aqui, não há nenhuma maneira de fazer o mesmo com um estilo de loop `for x in L:`, pois tal loop itera pelos itens reais e não por posições de uma lista. O loop `while` equivalente exige um pouco mais de trabalho de nossa parte:*

* Uma expressão de *compreensão de lista*, da forma `[x+1 for x in L]`, também faria um trabalho semelhante aqui, se bem que sem alterar a lista original no local (poderíamos atribuir o resultado do novo objeto lista da expressão de volta a `L`, mas isso não atualizaria quaisquer outras referências à lista original). Consulte o Capítulo 14 para ver mais informações sobre compreensões de lista.

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

Varreduras paralelas: zip e map

O truque da função `range` percorre seqüências com um loop `for` de maneira exaustiva. A função interna `zip` nos permite usar loops `for` para visitar várias seqüências em *paralelo*. Na operação básica, a função `zip` pega uma ou mais seqüências e retorna uma lista de tuplas que dispõem em pares os itens paralelos extraídos de seus argumentos. Por exemplo, suponha que estejamos trabalhando com duas listas:

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]
```

Para combinar os itens dessas listas, podemos usar a função `zip`:

```
>>> zip(L1,L2)
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Esse resultado pode ser útil em outros contextos. Contudo, quando aliado ao loop `for`, ele suporta iterações paralelas:

```
>>> for (x,y) in zip(L1, L2):
...     print x, y, '--', x+y
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

Aqui, percorremos o resultado da chamada de zip – os pares de itens extraídos das duas listas. Esse loop `for` usa atribuição de tupla novamente, para desempacotar cada tupla no resultado da função `zip` (na primeira vez, é como se executássemos `(x,y) =(1,5))`. O efeito é que percorremos `L1` e `L2` em nosso loop. Poderíamos obter um efeito semelhante com um loop while que manipulasse a indexação manualmente, mas teríamos mais coisas para digitar e poderia ser mais lento do que a estratégia `for/zip`.

A função `zip` é mais geral do que esse exemplo sugere. Por exemplo, ela aceita qualquer tipo de seqüência e mais de dois argumentos:

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> zip(T1,T2,T3)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

A função `zip` trunca as tuplas resultantes no comprimento da seqüência mais curta, quando os comprimentos dos argumentos diferem:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
```

```
>>> zip(S1, S2)
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

A relacionada e mais antiga função interna `map` cria pares de itens de seqüências de maneira semelhante, mas preenche as seqüências mais curtas com `None`, caso os comprimentos dos argumentos sejam diferentes:

```
>>> map(None, S1, S2)
[('a','x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
```

Na verdade, o exemplo está usando uma forma degenerada da função interna `map`. Normalmente, a função `map` recebe uma função, um ou mais argumentos de seqüência e reúne os resultados da chamada da função com itens paralelos extraídos das seqüências. Quando o argumento da função é `None` (como aqui), ela simplesmente cria pares de itens, como a função `zip`. A função `map` e ferramentas baseadas em função semelhantes serão abordadas no Capítulo 14.

Construção de dicionário com zip

Os dicionários sempre podem ser criados escrevendo um literal de dicionário ou pela atribuição de chaves com o passar do tempo:

```
>>> D1 = {'spam':1, 'eggs':3, 'toast':5}
>>> D1
{'toast': 5, 'eggs': 3, 'spam': 1}

>>> D1 = {}
>>> D1['spam'] = 1
>>> D1['eggs'] = 3
>>> D1['toast'] = 5
```

Contudo, o que fazer se seu programa recebe chaves e valores de dicionário em *listas* em tempo de execução, após você ter escrito seu script?

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]
```

Uma solução para ir das listas para um dicionário é usar a função `zip` nas listas e percorrê-las em paralelo com um loop `for`:

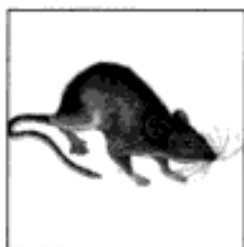
```
>>> zip(keys,vals)
[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'toast': 5, 'eggs':3, 'spam': 1}
```

Contudo, verifica-se que, no Python 2.2, você pode pular completamente o loop `for` e simplesmente passar as listas de chaves/valores da função `zip` para a chamada de construtor interno `dict`:

```
>>> keys = ['spam','eggs', 'toast']
>>> vals = [1, 3, 5]
>>> D3 = dict(zip(keys,vals))
>>> D3
{'toast': 5, 'eggs':3, 'spam': 1}
```

O nome interno `dict` é, na verdade, um nome de tipo no Python. Às vezes, chamá-lo é algo como uma conversão de lista para dicionário, mas na verdade é um pedido de construção de objeto (mais informações sobre nomes de tipo aparecem no Capítulo 23). Além disso, no Capítulo 14, conheceremos um conceito relacionado, porém mais rico: a *compreensão de lista*, que constrói listas em uma única expressão.

11

Documentando Código Python

Este capítulo conclui a Parte III com um exame das técnicas e ferramentas usadas para documentar código Python. Embora o código Python seja projetado para ser fácil de ler em geral alguns comentários bem colocados e legíveis para seres humanos podem ajudar outras pessoas a entenderem o funcionamento de seus programas. Para suportar comentários, o Python possui sintaxe e ferramentas para tornar a documentação mais fácil. Embora esse seja um conceito relacionado às ferramentas, o assunto é apresentado aqui, em parte porque envolve o modelo de sintaxe do Python e também como um recurso para os leitores que estejam lutando para entender o conjunto de ferramentas da linguagem. Como sempre, este capítulo termina com armadilhas e exercícios.

O ENTREATO DA DOCUMENTAÇÃO DO PYTHON

Neste ponto do livro, você provavelmente está começando a perceber que o Python vem com uma quantidade impressionante de funcionalidades previamente construídas – funções internas, exceções, atributos de objeto predefinidos, módulos de biblioteca padrão e muito mais. Além disso, apenas arranhamos a superfície de cada uma dessas categorias.

Uma das primeiras perguntas que os iniciantes desnorteados freqüentemente fazem é: como eu encontro informações sobre todas as ferramentas internas? Esta seção fornece dicas sobre várias fontes de documentação disponíveis no Python. Ela também apresenta as strings de documentação e o sistema *PyDoc*, que faz uso delas. Esses assuntos são um tanto periféricos em relação à linguagem básica em si, mas se tornam um conhecimento essencial assim que seu código atinge o nível dos exemplos e exercícios deste capítulo.

Fontes de documentação

Conforme resumido na Tabela 11-1, existe uma variedade de lugares para procurar informações no Python, com um volume de dados geralmente cada vez maior. Como a documentação é uma ferramenta fundamental na programação prática, vamos ver cada uma dessas categorias.

Tabela 11-1 Fontes de documentação do Python

Forma	Função
Comentários #	Documentação em arquivo
A função dir	Lista dos atributos disponíveis em objetos
Docstrings: `__doc__`	Documentação em arquivo vinculada aos objetos
PyDoc: a função de ajuda	Ajuda interativa para objetos
PyDoc: relatórios em HTML	Documentação de módulo em um navegador
Conjunto de manuais padrão	Descrições oficiais da linguagem e da biblioteca
Recursos na Web	Exercícios dirigidos on-line,exemplos etc.
Livros publicados	Textos de referência disponíveis comercialmente

Comentários

Os comentários com sinal # são a maneira mais básica de documentar seu código. Todo o texto após um sinal # (que não esteja dentro de uma literal de string) é simplesmente ignorado pelo Python. Por isso, esse é um lugar para você escrever e ler palavras significativas para os programadores. Tais comentários só são acessíveis em seus arquivos-fonte. Para escrever comentários mais amplamente disponíveis, use docstrings.

A função dir

A função interna `dir` é uma maneira fácil de ver uma lista que mostra todos os atributos disponíveis dentro de um objeto (isto é, seus métodos e itens de dados simples). Ela pode ser chamada com qualquer objeto que tenha atributos. Por exemplo, para descobrir o que está disponível no módulo `sys` da biblioteca padrão, importe-o e passe para a função `dir`:

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'__stderr__', '__stdin__', '__stdout__', '_getframe', 'argv',
'builtin_module_names', 'byteorder', 'copyright', 'displayhook', 'dlhandle',
'exc_info', 'exc_type', 'excepthook',
...mais nomes omitidos...]
```

Apenas alguns dos muitos nomes são mostrados. Execute essas instruções em sua máquina para ver a lista completa. Para descobrir quais atributos são fornecidos em tipos de objeto internos, execute a função `dir` em um literal desse tipo. Por exemplo, para ver atributos de lista e string, você pode passar objetos vazios:

```
>>> dir([ ])
['__add__', '__class__', ...more...
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']

>>> dir(' ')
['__add__', '__class__', ...more...
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'index', 'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
...mais nomes omitidos...]
```

Os resultados de dir para tipos internos incluem um conjunto de atributos relacionados à implementação do tipo (tecnicamente, métodos de sobrecarga de operador); todos eles começam e terminam com sublinhados duplos para diferenciá-los e podem ser seguramente ignorados neste ponto do livro, de modo que não são mostrados aqui.

A propósito, você pode obter o mesmo efeito passando um nome de tipo para a função dir, em vez de um literal:

```
>>> dir(str) == dir(' ')        # O mesmo resultado do exemplo anterior
1
>>> dir(list) == dir([ ])
1
```

Isso funciona porque funções como str e list, que já foram conversoras de tipo, na verdade são nomes de tipos; o fato de chamá-las ativa seus construtores para gerar uma instância desse tipo. Mais informações sobre construtores e métodos de sobrecarga de operador aparecerão quando conhecermos as classes, na Parte VI.

A função dir serve como um tipo de lembrete – ela fornece uma lista de nomes de atributo, mas não diz nada sobre o que esses nomes significam. Para obter tais informações extras, precisamos passar para o próximo assunto.

Docstrings: __doc__

Além dos comentários #, o Python suporta documentação que é mantida em tempo de execução para inspeção e a vincula automaticamente aos objetos. Sintaticamente, esses comentários são escritos como strings no início de arquivos de módulo e no início de instruções de função e de classe, antes de qualquer outro código executável. O Python preenche automaticamente a string, conhecida como *docstring*, no atributo __doc__ do objeto correspondente.

Docstrings definidas pelo usuário

Considere, por exemplo, o arquivo a seguir, *docstring.py*. Sua docstring aparece no início do arquivo e no início de uma função e de uma classe dentro dele. Aqui, usamos strings de bloco com aspas triplas para comentários de várias linhas no arquivo e na função, mas qualquer tipo de string funcionará. Ainda não estudamos as instruções def e class; portanto, ignore tudo a respeito delas, exceto as strings em seus inícios:

```
"""
Documentação do módulo
As palavras ficam aqui
"""

spam = 40
def square(x):
    """
    Documentação da função
    podemos ter seu fígado então?
    """
    return x **2

class employee:
    "documentação da classe"
    pass
```

```
print square(4)
print square.__doc__
```

A característica desse protocolo de documentação é que seus comentários são mantidos para inspeção em atributos `__doc__`, após o arquivo ser importado:

```
>>> import docstrings
16

    documentação da função
    podemos ter seu fígado então?

>>> print docstrings.__doc__

Documentação do módulo
As palavras ficam aqui

>>> print docstrings.square.__doc__

    documentação da função
    podemos ter seu fígado então?

>>> print docstrings.employee.__doc.__
documentação da classe
```

Aqui, após importar, exibimos as docstrings associadas ao módulo e seus objetos, imprimindo seus atributos `__doc__` onde o Python salvou o texto. Note que, geralmente, você desejará escrever `print` explicitamente para docstrings; caso contrário, obterá uma string simples com caracteres de nova linha incorporados.

Você também pode vincular docstrings a métodos de classes (abordados posteriormente), mas como elas são apenas instruções `def` aninhadas em uma classe, não são um caso especial. Para buscar a docstring de uma função de método no interior de uma classe, dentro de um módulo, siga o caminho e vá até a classe: `módulo.classe.método.__doc__` (veja o exemplo de docstring de método no Capítulo 22).

Padrões de docstring

Não há nenhum padrão geral sobre o que deve entrar no texto de uma docstring (embora algumas empresas tenham padrões internos). Existem várias propostas de linguagem e modelo de marcação (por exemplo, HTML), mas elas parecem não ter atraído a atenção no mundo Python.

Isso provavelmente está relacionado com a prioridade da documentação entre os programadores em geral. Normalmente, se você receber comentários em um arquivo, pode se considerar sortudo; pedir aos programadores para escrever manualmente seus comentários em HTML ou em outros formatos, parece improvável que dê certo. Naturalmente, o estimulamos a documentar seu código livremente.

Docstrings incorporadas

Verifica-se que os módulos internos e objetos do Python usam técnicas semelhantes para vincular documentação muito além das listas de atributos retornadas pela função `dir`. Por exemplo, para ver as palavras reais que fornecem uma descrição de um módulo interno legível para seres humanos, importe e imprima sua string `__doc__`:

```
>>> import sys
>>> print sys.__doc__
This module provides access to some objects
used or maintained by the interpreter and to
...mais texto omitido...

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
...mais texto omitido...
```

Analogamente, as funções, classes e métodos dentro dos módulos internos também têm palavras anexadas em seus atributos `__doc__`:

```
>>> print sys.getrefcount.__doc__
getrefcount(object) -> integer

Return the current reference count for the object.
...mais texto omitido...
```

Além disso, você pode ler sobre as funções internas por meio de suas docstrings:

```
>>> print int.__doc__
int(x[, base]) -> integer

Convert a string or number to an integer, if possible.
...mais texto omitido...

>>> print open.__doc__
file(name[, mode[, buffering]]) -> file object

Open a file. The mode can be 'r', 'w' or 'a' for reading
...mais texto omitido...
```

PyDoc: a função de ajuda

A técnica das docstrings se mostrou tão útil que o Python vem com uma ferramenta que as torna ainda mais fáceis de exibir. A ferramenta padrão PyDoc é um código Python que sabe extrair e formatar suas docstrings, junto com informações estruturais extraídas automaticamente, em relatórios de vários tipos e bem organizados.

Existem várias maneiras de chamar a ferramenta PyDoc, incluindo opções de script de linha de comando. Talvez as duas interfaces mais proeminentes da PyDoc sejam a função interna `help` e a interface de GUI/HTML. A recentemente introduzida função `help` ativa a PyDoc para gerar um relatório em texto simples (muito parecido com uma página de manual – manpage – em sistemas do tipo Unix):

```
>>> import sys
>>> help(sys.getrefcount)
Help on built-in function getrefcount:

getrefcount(...)
    getrefcount(object) -> integer
```

```
    Return the current reference count for the object.
    ...mais texto omitido...
```

Note que você não precisa importar sys para chamar help, mas precisa importar para obter ajuda sobre sys. Para objetos maiores, como módulos e classes, a tela da função help é dividida em várias seções, algumas das quais são mostradas aqui. Execute isso interativamente para ver o relatório completo.

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

DESCRIPTION
    This module provides access to some objects used
    or maintained by the interpreter and to functions
    ...mais...

FUNCTIONS
    __displayhook__ = displayhook(...)
        displayhook(object) -> None

        Print an object to sys.stdout and also save it
    ...mais...

DATA
    __name__ = 'sys'
    __stderr__ = <open file '<stderr>', mode 'w' at 0x0082BEC0>
    ...mais...
```

Algumas das informações desse relatório são docstrings e algumas (por exemplo, padrões de chamada de função) são dados estruturais que o Python compila automaticamente, inspecionando os detalhes internos dos objetos. Você também pode usar a função help em funções, métodos e tipos internos. Para obter ajuda sobre um tipo interno, use o nome do tipo (por exemplo, dict para dicionário, str para string, list para lista); você obterá uma tela grande, descrevendo todos os métodos disponíveis para esse tipo:

```
>>> help(dict)
Help on class dict in module __builtin__:

class dict(object)
 | dict() -> new empty dictionary.
 ...mais...

>>> help(str.replace)
Help on method_descriptor:

replace(...)
    S.replace (old, new[, maxsplit]) -> string

    Return a copy of string S with all occurrences
    ...mais...
```

```
>>> help(ord)
Help on built-in function ord:

ord(...)
    ord(c) -> integer

    Return the integer ordinal of a one-character string.
```

Finalmente, a função `help` funciona em seus módulos tão bem quanto nos internos. Aqui está o relatório sobre o arquivo `docstrings.py` escrito na seção anterior; novamente, parte disso são docstrings e parte é automático da estrutura:

```
>>> help(docstrings.square)
Help on function square in module docstrings:

square(x)
    Documentação da função
    podemos ter seu fígado então?

>>> help(docstrings.employee)
...mais...

>>> help (docstrings)
Help on module docstrings:

NAME
    docstrings

FILE
    c:\python22\docstrings.py

DESCRIPTION
    Documentação do módulo
    As palavras ficam aqui

CLASSES
    employee
    ...mais...

FUNCTIOS
    square(x)
        Documentação da função
        podemos ter seu fígado então?

DATA
    __file__ = 'C:\\PYTHON22\\docstrings.pyc'
    __name__ = 'docstrings'
    spam = 40
```

PyDoc: relatórios em HTML

A função `help` é ótima para capturar documentação ao se trabalhar interativamente. Para exibir uma tela mais grandiosa, a ferramenta PyDoc também fornece uma interface de GUI (um script Python/TKinter simples, porém portável) e pode exibir seu relatório em formato de página HTML, visível em qualquer navegador da Web. Nesse modo, a ferramenta PyDoc pode

Figura 11-1 Interface de pesquisa de nível superior da GUI da ferramenta PyDoc.

ser executada de forma local ou como um servidor remoto, e os relatórios contêm hyperlinks que permitem clicar na documentação de componentes relacionados em seu aplicativo.

Para iniciar a ferramenta PyDoc nesse modo, geralmente você primeiro chama a GUI do mecanismo de pesquisa capturado na Figura 11-1. Você pode iniciá-lo selecionando o item

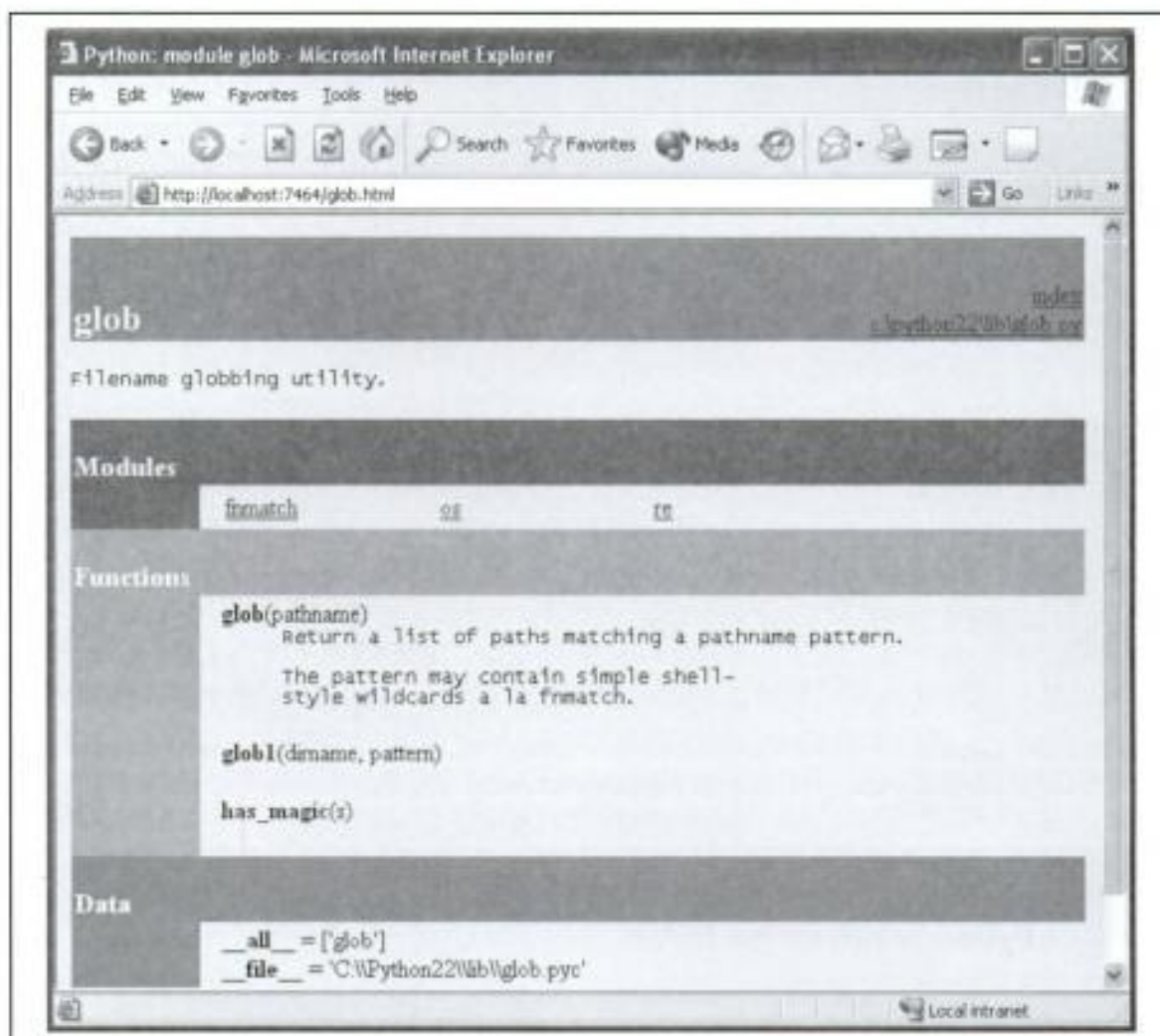



Figura 11-2 Relatório em HTML do PyDoc, módulo interno.

Module Docs, no menu do botão Iniciar do Python no Windows, ou chamar o script `pydocgui` no diretório de ferramentas do Python. Digite o nome de um módulo sobre o qual você esteja interessado em conhecer e pressione a tecla Enter. A ferramenta PyDoc percorrerá o caminho de pesquisa de importação do seu módulo, procurando referências para ele.

Quando você tiver encontrado uma entrada interessante, selecione e clique em "go to selected". A ferramenta PyDoc ativa um navegador da Web em sua máquina para exibir o relatório representado no formato HTML. A Figura 11-2 mostra as informações que a ferramenta PyDoc exibe para o módulo interno `glob`.

Observe os hyperlinks na seção Module dessa página – dê um clique neles para ir a essas páginas do PyDoc de módulos relacionados (importados). Para páginas maiores, a ferramenta PyDoc também gera hyperlinks para seções dentro da página. Assim como a interface da função `help`, a interface de GUI também funciona em módulos definidos pelo usuário. A Figura 11-3 mostra a página gerada para o arquivo de módulo *docstrings.py*.

A ferramenta PyDoc pode ser personalizada e chamada de várias maneiras. O mais importante a ser aprendido nesta seção é que, basicamente, a ferramenta PyDoc fornece relatórios de implementação "gratuitamente" – se você souber usar bem docstrings em seus arquivos,

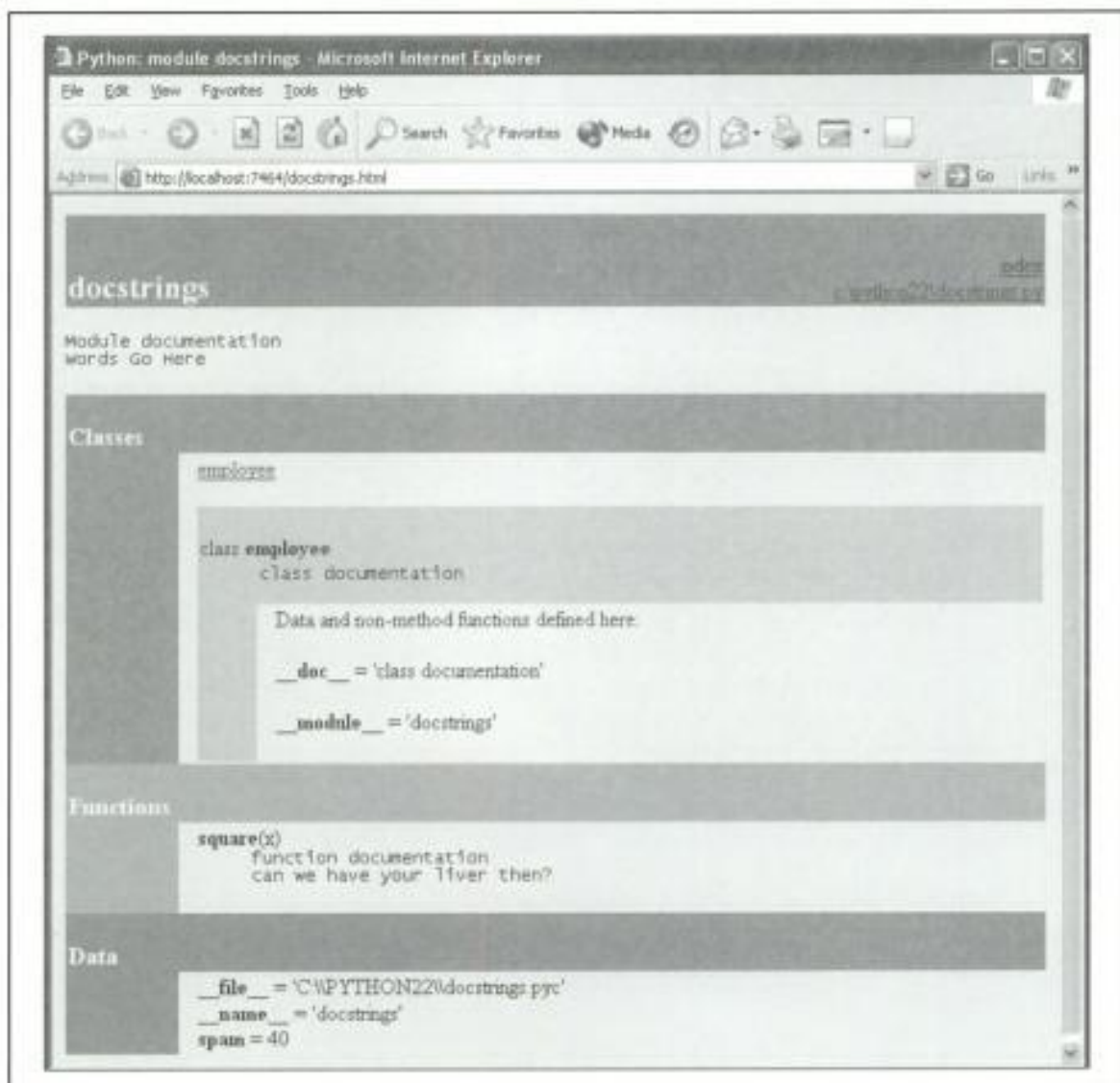



Figura 11-3 Relatório em HTML do PyDoc, módulo definido pelo usuário.

o PyDoc fará todo o trabalho de coletá-los e formatá-los para exibição. A ferramenta PyDoc também fornece uma maneira fácil de acessar um nível médio de documentação para ferramentas internas – seus relatórios são mais úteis do que as listas de atributos brutas e menos exaustivos do que os manuais padrão.

Conjunto de manuais padrão

Para ver a descrição completa e mais atualizada da linguagem Python e seu conjunto de ferramentas, os manuais padrão da linguagem estão prontos para ser usados. Os manuais vêm em formato HTML e são instalados com o sistema Python no Windows – eles estão disponíveis no menu do botão Iniciar do Python, e também podem ser abertos a partir do menu Help dentro do IDLE. Você também pode buscar o conjunto de manuais separadamente, no endereço *http://www.python.org*, em uma variedade de formatos, ou lê-los online nesse site (siga o link Documentation).

Quando aberto, o formato HTML dos manuais exibe uma página-raiz, como a que aparece na Figura 11-4. As duas entradas mais importantes aqui provavelmente são Library Reference (que documenta os tipos internos, funções, exceções e módulos de biblioteca padrão) e Language Re-

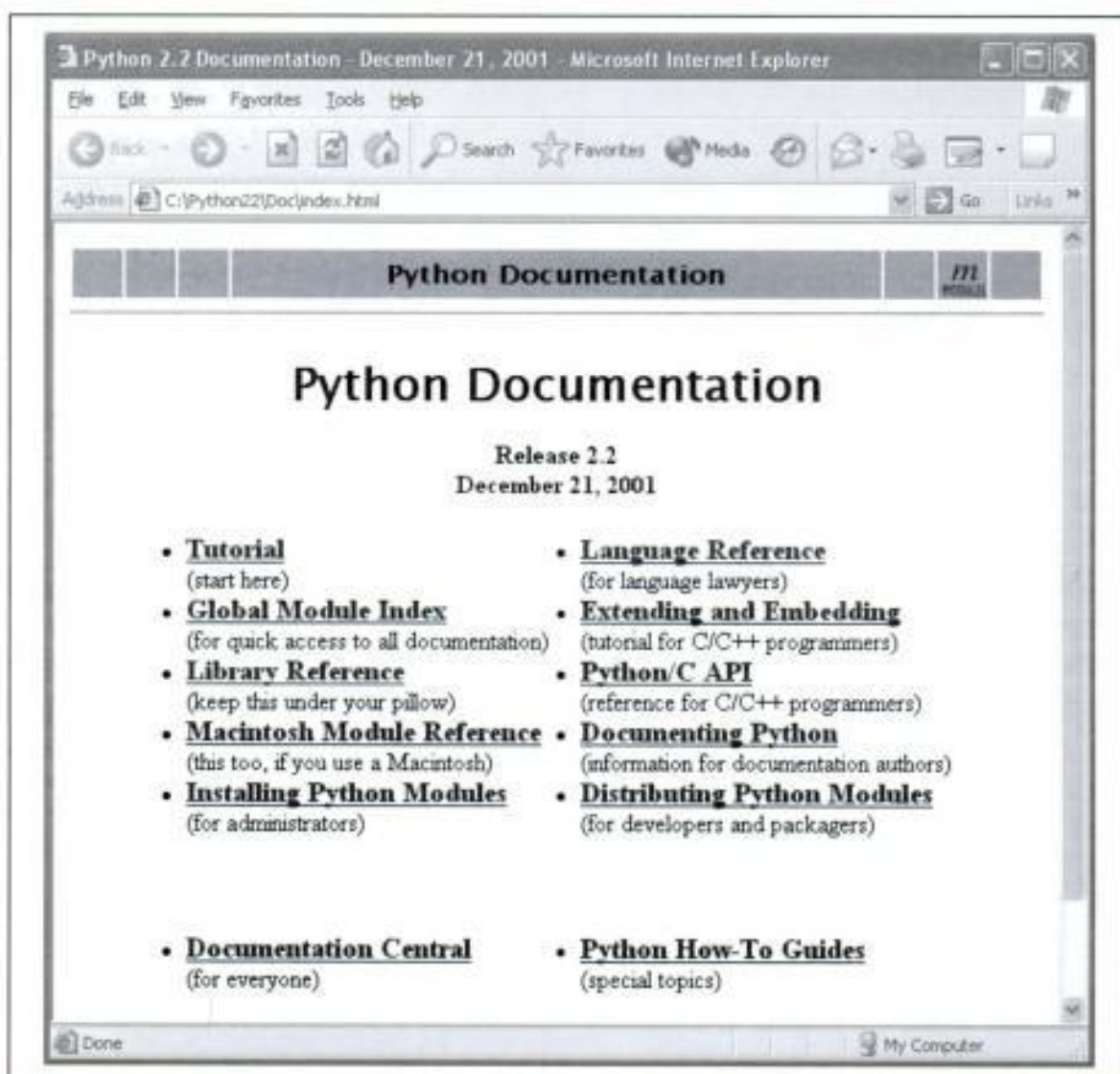



Figura 11-4 Conjunto de manuais padrão do Python.

ference (que fornece uma descrição formal dos detalhes em nível de linguagem). O exercício dirigido (tutorial) listado nessa página também fornece uma breve introdução para os iniciantes.

Recursos na Web

No endereço *http://www.python.org*, você encontrará links para vários exercícios dirigidos, alguns dos quais abordam tópicos ou domínios especiais. Procure os links Documentation e Newbies (isto é, iniciantes). Esse site também lista recursos do Python em outros idiomas que não o inglês.

Livros publicados

Finalmente, hoje você pode escolher uma coleção de livros de referência sobre o Python. Em geral, os livros tendem a não acompanhar as últimas alterações do Python, parcialmente por causa do trabalho envolvido na escrita e parcialmente devido aos atrasos naturais incorporados ao ciclo de publicação. Normalmente, quando um livro é publicado, ele está seis meses ou mais atrás do estado atual do Python. Ao contrário dos manuais padrão, geralmente os livros também não são gratuitos.

Para muitas pessoas, a conveniência e a qualidade de um texto publicado profissionalmente, vale seu custo. Além disso, o Python muda tão lentamente que, normalmente, os livros ainda são relevantes anos após serem publicados, especialmente se seus autores divulgam atualizações na Web. Consulte o Prefácio para mais indicações sobre livros de Python.

PROBLEMAS DE DESENVOLVIMENTO COMUNS

Antes dos exercícios de programação desta parte do livro, aqui estão alguns dos erros mais comuns que os iniciantes cometem ao escrever instruções e programas em Python. Você vai aprender a evitá-los quando tiver obtido um pouco de experiência no desenvolvimento em Python, mas algumas palavras podem ajudá-lo a não cair em algumas dessas armadilhas inicialmente.

Não se esqueça dos dois-pontos. Não se esqueça de digitar : no final de cabeçalhos de instruções compostas (a primeira linha de uma instrução `if`, `while`, `for` etc.). De qualquer modo, no início você provavelmente vai esquecer (nós também esquecemos), mas pode se confortar pelo fato de que em breve isso se tornará um hábito inconsciente.

Comece na coluna 1. Certifique-se de começar o código de nível superior (não aninhado) na coluna 1. Isso inclui código não aninhado digitado em arquivos de módulo, assim como código não aninhado digitado no prompt interativo.

As linhas em branco importam no prompt interativo. As linhas em branco de instruções compostas são sempre ignoradas em arquivos de módulo, mas ao se digitar código, finalizam a instrução no prompt interativo. Em outras palavras, as linhas em branco informam à linha de comando interativa que você terminou uma instrução composta. Se você quiser continuar, não pressione a tecla Enter no prompt..., até ter realmente terminado.

Faça a endentação consistentemente. Evite misturar tabulações e espaços na endentação de um bloco, a não ser que você saiba o que seu sistema editor de textos fará com as tabulações. Caso contrário, o que você vê em seu editor pode não ser o que o Python vê ao contar tabulações como um número de espaços. É mais seguro usar apenas tabulações ou apenas espaços para cada bloco.

Não escreva código C em Python. Uma nota para programadores de C/C++: você não precisa digitar parênteses em torno de testes em cabeçalhos `if` e `while` (por exemplo, `if (X==1):`). Se quiser fazer isso, você pode (qualquer expressão pode ser colocada entre parênteses), mas eles são totalmente supérfluos nesse contexto. Além disso, não termine todas as suas instruções com um ponto-e-vírgula. Tecnicamente, também é válido fazer isso no Python, mas é totalmente inútil, a menos que você esteja colocando mais de uma instrução em uma única linha (o final de uma linha normalmente finaliza uma instrução). Lembre-se de não incorporar instruções de atribuição em testes de loop `while` e não use `{}` em torno de blocos (em vez disso, endente consistentemente seus blocos de código aninhados).

Use loops `for` simples, em vez de `while` ou `range`. Um loop `for` simples (por exemplo, `for x in seq:`) quase sempre é mais fácil de escrever e mais rápido para executar do que um loop contador baseado em `while` ou `range`. Como o Python manipula a indexação internamente para um loop `for` simples, às vezes ele pode ser duas vezes mais rápido do que o loop `while` equivalente.

Não espere resultados de funções que alteram objetos no local. As operações de alteração no local, como os métodos `list.append()` e `list.sort()` do Capítulo 6, não retornam um valor (a não ser `None`). Chame-as sem atribuir o resultado. Não é incomum iniciantes escreverem algo como `mylist=mylist.append(X)` para tentar obter o resultado de uma instrução `append`. Em vez disso, essa construção atribui `None` a `mylist`, em vez de modificar a lista (na verdade, você perderá completamente a referência para a lista).

Um exemplo mais tortuoso disso aparece ao se tentar percorrer itens de dicionário de maneira ordenada. É muito comum ver este tipo de código: `for k in D.keys().sort():`. Isso quase funciona: o método `keys` constrói uma lista de chaves e o método `sort` a ordena, mas como o método `sort` retorna `None`, o loop falha, pois, em última análise, é um loop sobre `None` (não uma seqüência). Para escrever isso corretamente, divida as chamadas de método em instruções: `Ks = D.keys()`, em seguida, `Ks.sort()` e, finalmente, `for k in Ks:`. A propósito, esse é um caso em que você ainda desejará chamar o método `keys` explicitamente para fazer o loop, em vez de contar com os iteradores de dicionário.

Sempre use parênteses para chamar uma função. Você deve adicionar parênteses após um nome de função para chamá-la, receba ela argumento ou não (por exemplo, use `função()` e não `função`). Na Parte IV, veremos que as funções são simplesmente objetos que têm uma operação especial – uma chamada, que você dispara com os parênteses.

Nas aulas, isso parece ocorrer mais freqüentemente com arquivos. É comum ver iniciantes digitarem `arquivo.close` para fechar um arquivo, em vez de `arquivo.close()`. Como é válido referenciar uma função sem chamá-la, a primeira versão sem parênteses funciona silenciosamente, mas não fecha o arquivo!

Não use extensões nem caminhos em importações e recarregamentos. Omita caminhos de diretório e sufixos de arquivo em instruções de importação (por exemplo, escreva `import mod` e não `import mod.py`). (Conhecemos os fundamentos dos módulos no Capítulo 6 e continuaremos a estudá-los na Parte V.) Como os módulos podem ter outros sufixos, além de *.py* (*.pyc*, por exemplo), incorporar o código de um sufixo em particular não é apenas uma sintaxe inválida, como também não faz sentido. Além disso, a sintaxe de caminho de diretório, específica da plataforma, vem das configurações do caminho de pesquisa do módulo e não da instrução de importação.

EXERCÍCIOS DA PARTE III

Agora que você já sabe desenvolver lógica de programa básico, os exercícios pedem para que implemente algumas tarefas simples com instruções. A maior parte do trabalho está no exercício 4, que permite explorar alternativas de código. Sempre existem muitas maneiras de organizar as instruções, e parte do aprendizado do Python é conhecer quais organizações funcionam melhor do que as outras.

1. *Desenvolvimento de loops básicos.*
   a. Escreva um loop `for` que imprima o código ASCII de cada caractere de uma string chamada `S`. Use a função interna `ord(caractere)` para converter cada caractere em um inteiro em ASCII. (Teste isso interativamente para ver como funciona.)
   b. Em seguida, altere seu loop para calcular a soma dos códigos ASCII de todos os caracteres de uma string.
   c. Finalmente, modifique seu código novamente, para retornar uma nova lista contendo os códigos ASCII de cada caractere da string. Esta expressão tem um efeito semelhante – `map(ord,S)`? (Dica: consulte a Parte IV.)
2. *Caracteres de barra invertida.* O que acontece em sua máquina quando você digita interativamente o código a seguir?

```
for i in range(50):
    print 'hello %d\n\a' % i
```

   Tenha cuidado, pois se você executar fora da interface IDLE, este exemplo pode fazer soar um bip; portanto, talvez você não queira executá-lo em um laboratório cheio de gente. Em vez disso, o IDLE imprime caracteres estranhos (veja os caracteres de escape de barra invertida na Tabela 5-2).
3. *Ordenação de dicionários.* No Capítulo 6, vimos que os dicionários são coleções desordenadas. Escreva um loop `for` que imprima os itens de um dicionário em ordem (ascendente). Dica: use os métodos de dicionário `keys` e de lista `sort`.
4. *Alternativas de lógica de programa.* Considere o código a seguir, o qual usa um loop `while` e o flag `found` para pesquisar uma lista de potências de `2`, em busca do valor `2` elevado à 5ª potência (32). Ele está armazenado em um arquivo de módulo chamado *power.py*.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = i = 0
while not found and i < len(L):
    if 2 ** X == L[i]:
        found = 1
    else:
        i = i+1

if found:
    print 'at index', i
else:
    print X, 'not found'
C:\book\tests> python power.py
at index 5
```

Como está, o exemplo não segue as técnicas de desenvolvimento normais do Python. Siga os passos abaixo para aprimorá-lo. Para todas as transformações, você pode digitar seu código interativamente ou armazená-lo em um arquivo de script executado a partir da linha de comando do sistema (usar um arquivo torna este exercício muito mais fácil).

a. Primeiro, reescreva esse código com uma cláusula `else` de loop `while`, para eliminar o flag `found` e a instrução `if` final.

b. Em seguida, reescreva o exemplo para usar um loop `for` com uma cláusula `else`, para eliminar a lógica de indexação de lista explícita. Dica: para obter o índice de um item, use o método de lista `index` (`L.index(X)` retorna o deslocamento do primeiro `X` na lista `L`).

c. Então, remova o loop completamente, reescrevendo os exemplos com uma expressão simples de participação como membro com o operador `in`. (Consulte o Capítulo 6 para ver mais detalhes ou digite o seguinte para testar: `2 in [1,2,3]`.)

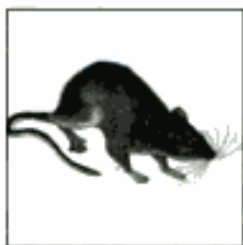
d. Finalmente, use um loop `for` e o método de lista `append` para gerar a lista de potências de `2 (L)`, em vez de incorporar o código de uma literal de lista.

e. Pensamentos mais profundos: (1) Você acha que mover a expressão `2**X` para fora dos loops melhoraria o desempenho? Como você escreveria isso? (2) Conforme vimos no exercício 1, o Python também inclui uma ferramenta `map(função, lista)` que pode gerar a lista de potências de 2: `map(lambda x: 2**x, range(7))`. Experimente digitar esse código interativamente. Conheceremos a função `lambda` mais formalmente no Capítulo 14.

IV

Funções

Na Parte IV, estudaremos as funções do Python – pacotes de código que podem ser chamados repetidamente, com diferentes entradas e saída a cada vez. Já chamamos funções anteriormente no livro: `open`, para fazer um objeto arquivo, por exemplo. Aqui, o enfoque será o desenvolvimento de funções definidas pelo usuário, as quais calculam valores, executam parte da lógica global de um programa ou englobam código de alguma forma para fácil reutilização.

12

Fundamentos das Funções

Na Parte III, vimos as instruções procedurais básicas do Python. Aqui, passaremos a explorar um conjunto de instruções adicionais que criam suas próprias funções. Em termos simples, uma função é um dispositivo que agrupa um conjunto de instruções, de modo que elas possam ser executadas mais de uma vez em um programa. As funções também nos permitem especificar parâmetros que servem como entradas e podem diferir a cada vez que o código de uma função é executado. A Tabela 12-1 resume as principais ferramentas relacionadas às funções que estudaremos nesta parte do livro.

Tabela 12-1 Instruções e expressões relacionadas às funções

Instrução	Exemplos
Chamadas	`myfunc("spam", ham, "toast")`
`def, return, yield`	`def adder(a, b=1 *c): return a+b+c[0]`
`global`	`def function(): global x; x = 'new'`
`lambda`	`funcs = [lambda x: x**2, lambda x: x*3]`

POR QUE USAR FUNÇÕES?

Antes de entrarmos nos detalhes, vamos ver um quadro claro do que são as funções. As funções são um dispositivo de estruturação de programas, quase universal. A maior parte de vocês provavelmente já as encontrou em outras linguagens, onde elas podem ter sido chamadas de sub-rotinas ou procedures. Mas, como uma breve introdução, as funções têm duas tarefas principais no desenvolvimento:

Reutilização de código

Assim como na maioria das linguagens de programação, as funções do Python são a maneira mais simples de empacotar lógica que você precisa usar em mais de um lugar e mais de uma vez. Até agora, todo código que escrevemos era executado imediatamente. As funções nos permitem agrupar e generalizar código para ser usado arbitrariamente muitas vezes posteriormente.

Decomposição procedural

As funções também fornecem uma ferramenta para dividir os sistemas em partes, com tarefas bem definidas. Por exemplo, para fazer uma pizza desde o início, você começaria misturando a massa, enrolando-a, adicionando os ingredientes, assando etc. Se você fosse programar um robô pizzaiolo, as funções o ajudariam a dividir a tarefa global "fazer pizza" em partes – uma função para cada sub-tarefa do processo. É mais fácil implementar as tarefas menores isoladamente do que implementar o processo inteiro de uma só vez. Em geral, as funções são como um procedimento – como fazer algo, em vez do que você está fazendo. Vamos ver por que essa distinção é importante, na Parte VI.

Nesta parte do livro, vamos explorar as ferramentas usadas para desenvolver funções no Python: fundamentos das funções, regras de escopo e passagem de argumentos, junto com alguns conceitos relacionados. Conforme veremos, as funções não significam muita sintaxe nova, mas nos levam a algumas idéias de programação maiores.

DESENVOLVENDO FUNÇÕES

Embora não tenhamos sido muito formais, já usamos funções em capítulos anteriores. Por exemplo, para fazer um objeto arquivo, chamamos a função interna `open`. Analogamente, usamos a função interna `len` para solicitar o número de itens em um objeto coleção.

Neste capítulo, aprenderemos a escrever *novas* funções em Python. As funções que escrevemos comportam-se da mesma maneira que as internas já vistas: elas são chamadas em expressões, recebem valores e retornam resultados. Mas escrever novas funções exige algumas noções adicionais que ainda não vimos aplicadas. Além disso, as funções se comportam de maneira muito diferente no Python, em relação ao que acontece em linguagens compiladas, como C. Aqui está uma breve introdução aos principais conceitos existentes por trás das funções do Python, os quais estudaremos neste capítulo:

`def` ***é código executável.*** As funções do Python são escritas com uma nova instrução, `def`. Ao contrário das funções em linguagens compiladas, como C, `def` é uma instrução executável – sua função não existe até que o Python busque e execute a instrução `def`. Na verdade, é válido (e até útil, ocasionalmente) aninhar instruções `def` dentro de instruções `if`, loops e até outras instruções `def`. Na operação típica, as instruções `def` são escritas em arquivos de módulo e, naturalmente, são executadas para gerar funções quando o arquivo de módulo é importado pela primeira vez.

`def` ***cria um objeto e atribui um nome a ele.*** Quando o Python busca e executa uma instrução `def`, ele gera um novo objeto função e atribui o nome da função a ele. Assim como acontece com todas as atribuições, o nome da função torna-se uma referência para o objeto função. Não há nada de mágico sobre o nome de uma função – conforme veremos, o objeto função pode receber outros nomes, ser armazenado em uma lista etc. As funções também podem ser criadas com a expressão `lambda` – um conceito mais avançado, deixado para depois neste capítulo.

`return` ***envia um objeto resultado de volta para quem fez a chamada*** Quando uma função é chamada, quem a chamou é interrompido até que ela termine seu trabalho e retorne o controle. As funções que calculam um valor, o enviam de volta para quem fez a chamada, com uma instrução `return`; o valor retornado torna-se o resultado da chamada de função. Funções conhecidas como geradoras também podem usar a instrução `yield` para enviar um valor de volta e suspender seu estado, para que possam ser retomadas posteriormente. Esse também é um assunto avançado, abordado posteriormente neste capítulo.

Os argumentos são passados por atribuição (referência de objeto). No Python, os argumentos são passados para as funções por atribuição (a qual, conforme aprendemos, significa referência de objeto). Conforme veremos, isso não é exatamente igual às regras de passagem da linguagem C ou aos parâmetros de referência da linguagem C++ – quem fez a chamada e a função compartilham objetos pelas referências, mas não há nenhum alias de nome. Alterar o nome de um argumento não altera também um nome em quem fez a chamada, mas a alteração de objetos mutáveis passados pode alterar os objetos compartilhados por quem fez a chamada.

`global` ***declara variáveis em nível de módulo que devem ser atribuídas.*** Por padrão, todos os nomes atribuídos em uma função são locais para essa função, e existem apenas enquanto a função é executada. Para atribuir um nome no módulo que as envolvem, as funções precisam listá-lo em uma instrução global. Em geral, os nomes são sempre pesquisados em *escopos* – lugares onde as variáveis são armazenadas – e atribuições vinculam nomes aos escopos.

Argumentos, valores de retorno e variáveis não são declarados. Assim como tudo no Python, não há nenhuma restrição de tipo nas funções. Na verdade, nada a respeito de uma função precisa ser declarado antecipadamente: podemos passar argumentos de qualquer tipo, retornar qualquer tipo de objeto etc. Como conseqüência, freqüentemente uma única função pode ser aplicada em uma variedade de tipos de objeto.

Se algumas das palavras anteriores não foram compreendidas, não se preocupe – vamos explorar todos esses conceitos com código real neste capítulo. Vamos começar expandindo essas idéias e examinando alguns exemplos iniciais pelo caminho.

Instruções def

A instrução `def` cria um objeto função e atribui um nome a ele. Sua forma geral é a seguinte:

```
def <nome>(arg1, arg2,... argN):
    <instruções>
```

Assim como acontece com todas as instruções compostas do Python, `def` consiste em uma linha de cabeçalho, seguida por um bloco de instruções, normalmente endentadas (ou uma instrução simples após os dois-pontos). O bloco de instruções torna-se o *miolo* da função – o código executado pelo Python sempre que a função é chamada. A linha de cabeçalho especifica um *nome* de função, que é atribuído ao objeto função, junto com uma lista de zero ou mais *argumentos* (às vezes chamados de parâmetros) entre parênteses. Os nomes de argumento no cabeçalho serão atribuídos aos objetos passados entre parênteses no momento da chamada.

O corpo das funções freqüentemente contém uma instrução `return`:

```
def <nome>(arg1, arg2,...argN):
    ...
    return <valor>
```

A instrução `return` do Python pode aparecer em qualquer lugar no miolo de uma função; ela finaliza a chamada de função e envia um resultado de volta para quem fez a chamada. Ela é composta de uma expressão de objeto que fornece o resultado da função. A instrução `return` é opcional; se não estiver presente, uma função termina quando o fluxo de controle chega no final do seu miolo. Tecnicamente, uma função sem uma instrução `return` retorna o objeto `None` automaticamente, mas normalmente ele é ignorado.

A instrução def é executada em tempo de execução

A instrução `def` do Python é verdadeiramente executável: ao ser executada, ela cria e atribui um nome ao novo objeto função. Como se trata de uma instrução, ela pode aparecer em qualquer lugar onde pode haver uma instrução – até mesmo aninhada em outras instruções. Por exemplo, é completamente válido aninhar uma função `def` dentro de uma instrução `if`, para selecionar entre definições alternativas:

```
if test:
   def func():              # Define func desta maneira.
       ...
else:
   def func():              # Ou, então, desta maneira.
       ...
...
func()                      # Chama a versão selecionada e construída.
```

Uma maneira de entender esse código é perceber que a instrução `def` é muito parecida com uma instrução `=`: ela simplesmente atribui um nome em tempo de execução. Ao contrário das linguagens compiladas, como C, as funções do Python não precisam ser totalmente definidas antes que o programa execute. Em geral, as instruções `def` não são avaliadas até serem buscadas e executadas, e o código *dentro* das instruções `def` não é avaliado até que a função seja chamada posteriormente.

Como a definição da função ocorre em tempo de execução, não há nada de especial sobre o nome da função, apenas o objeto a que se refere:

```
othername = func            # Atribui o objeto função.
othername()                 # Chama func novamente.
```

Aqui, a função recebeu um nome diferente e foi chamada por meio do novo nome. Assim como tudo no Python, as funções são apenas objetos; elas são gravadas explicitamente na memória, no momento da execução do programa.

UM PRIMEIRO EXEMPLO: DEFINIÇÕES E CHAMADAS

Fora os conceitos de tempo de execução (que tendem a parecer mais exclusivos dos programadores com experiência em linguagens compiladas tradicionais), as funções do Python são simples de usar. Vamos escrever um primeiro exemplo real para demonstrarmos os fundamentos. Na realidade, existem dois lados no quadro da função: uma *definição* – a instrução `def` que cria uma função – e uma *chamada* – uma expressão que diz ao Python para que execute o miolo da função.

Definição

Aqui está uma definição, digitada interativamente, que define uma função chamada `times`, a qual retorna o produto de seus dois argumentos:

```
>>> def times(x, y):        # Cria e atribui função.
...     return x * y        # Corpo executado quando chamado.
...
```

Quando o Python busca e executa essa instrução `def`, ele cria um novo objeto função que empacota o código da função e atribui ao objeto o nome `times`. Normalmente, essa instrução

é escrita em um arquivo de módulo e seria executada quando o arquivo que a contém fosse importado. Contudo, para algo tão pequeno assim, o prompt interativo é suficiente.

Chamadas

Após a execução da instrução `def`, o programa pode chamar (executar) a função adicionando parênteses após seu nome. Opcionalmente, os parênteses podem conter um ou mais argumentos de objeto, a serem passados (atribuídos) para os nomes no cabeçalho da função:

```
>>> times(2, 4)                # Argumentos entre parênteses
8
```

Essa expressão passa dois argumentos para `times`: o nome `x` no cabeçalho da função recebe o valor `2`, `y` recebe `4` e o miolo da função é executado. Neste caso, o miolo é apenas uma instrução `return`, a qual envia de volta o resultado, como o valor da expressão de chamada. O objeto retornado é impresso interativamente aqui (assim como na maioria das linguagens, `2*4` é `8` no Python). Ele também poderia ser atribuído a uma variável, se precisássemos usá-lo posteriormente:

```
>>> x = times(3.14, 4)         # Salva o objeto resultado.
>>> x
12.56
```

Agora, observe o que acontece quando a função é chamada uma terceira vez, com tipos de objetos muito diferentes passados:

```
>>> times('Ni', 4)                      # As funções são "sem tipo".
'NiNiNiNi'
```

Nessa terceira chamada, uma string e um inteiro são passados para `x` e `y`, em vez de dois números. Lembre-se de que `*` funciona tanto em números como em seqüências. Como você nunca declara os tipos de variáveis, argumentos ou valores de retorno, pode usar `times` para *multiplicar* números ou *repetir* seqüências.

Polimorfismo no Python

Na verdade, o significado real da expressão `x * y` na função `times` simples depende completamente dos tipos de objetos que `x e y` são – isso significa multiplicação primeiro e repetição depois. O Python deixa por conta dos *objetos* fazer algo razoável para essa sintaxe.

Esse tipo de comportamento, que dependendo do tipo é conhecido como *polimorfismo* (o significado das operações depende dos objetos que estão sendo utilizados). Como o Python é uma linguagem tipada dinamicamente, o polimorfismo corre solto: toda operação é polimórfica no Python.

Isso é algo deliberado e contribui muito para a flexibilidade da linguagem. Uma única função, por exemplo, geralmente pode ser aplicada a toda uma categoria de tipos de objeto. Contanto que esses objetos suportem a *interface* (também conhecida como protocolo) esperada, eles podem ser processados pela função. Isto é, se os objetos passados para uma função possuem os métodos e operadores esperados, eles são totalmente compatíveis com a lógica da função.

Mesmo em nossa função `times` simples, isso significa que *quaisquer* dois objetos que suportem um operador `*` funcionarão, independente do que possam ser e de quando possam ser escritos. Além disso, se os objetos passados *não* suportam essa interface esperada, o Python detectará o erro quando a expressão `*` for executada e lançará uma exceção automaticamente. É inútil desenvolvermos uma verificação de erro aqui.

Isso revela-se uma diferença filosófica fundamental entre o Python e as linguagens tipadas estaticamente, como C++ e Java: no Python, seu código *não é obrigado a se preocupar* com tipos de dados específicos. Se fosse, estaria limitado a funcionar apenas com os tipos antecipados por você ao escrever seu código. Ele não suportaria outros tipos de objeto compatíveis escritos no futuro. Embora seja possível testar os tipos com ferramentas como a função interna `type`, fazer isso prejudica a flexibilidade do seu código. De modo geral, desenvolvemos para *interfaces* de objeto no Python e não para tipos de dados.

UM SEGUNDO EXEMPLO: INTERSEÇÃO DE SEQÜÊNCIAS

Vamos ver um segundo exemplo de função que faz algo um pouco mais útil do que multiplicar argumentos e ilustra melhor os fundamentos das funções.

No Capítulo 10, vimos um loop `for` que reunia itens em comum de duas strings. Lá, observamos que o código não era tão útil quanto poderia ser, pois estava configurado para funcionar apenas com variáveis específicas e não podia ser novamente executado. Naturalmente, você poderia recortar e colar o código em cada lugar que ele precisasse ser executado, mas essa solução não é boa nem geral – você ainda teria que editar cada cópia para suportar diferentes nomes de seqüência e, então, alterar o algoritmo exigiria alterar várias cópias.

Definição

Agora, você provavelmente pode supor que a solução para esse dilema é empacotar o loop `for` dentro de uma função. As funções oferecem diversas vantagens em relação a um código de nível superior simples:

- Colocando o código em uma função, ele torna-se uma ferramenta que pode ser executada quantas vezes você quiser.
- Permitindo que quem fizer a chamada passe argumentos arbitrários, você a torna geral o suficiente para funcionar em quaisquer duas seqüências em que queira fazer a interseção.
- Empacotando a lógica em uma função, você só precisa alterar código em apenas um lugar, se precisar mudar o funcionamento da interseção.
- Escrevendo a função em um arquivo de módulo, ela pode ser importada e reutilizada por qualquer programa executado em sua máquina.

Na verdade, encerrar o código em uma função o transforma em um utilitário de interseção geral:

```
def intersect(seq1, seq2):
    res = []                        # Começa vazio.
    for x in seq1:                  # Percorre seq1.
        if x in seq2:               # Item comum?
            res.append(x)           # Adiciona no fim.
return res
```

A transformação do código simples do Capítulo 10 nessa função é simples e direta; apenas aninhamos a lógica original sob um cabeçalho `def` e tornamos os objetos nos quais ela opera nomes de parâmetro passados. Como essa função calcula um resultado, também adicionamos uma instrução `return` para enviar um objeto resultado de volta para quem fez a chamada.

Chamadas

Antes que você possa chamar a função, precisa fazê-la. Execute sua instrução `def` digitando-a interativamente ou escrevendo-a em um arquivo de módulo e importando o arquivo. Quando você tiver executado a instrução `def`, de uma maneira ou de outra, a função é chamada por meio da passagem de quaisquer dois objetos seqüência entre parênteses:

```
>>> s1 = "SPAM"
>>> s2 = "SCAM"

>>> intersect(s1, s2)           # Strings
['S', 'A', 'M']
```

Aqui, o código passa duas strings e recebe de volta uma lista contendo os caracteres em comum. O algoritmo utilizado pela função é simples: " para cada item no primeiro argumento, se esse item também está no segundo argumento, anexe o item no resultado". É um pouco mais curto escrever isso em Python do que em português, mas funciona da mesma forma.

Polimorfismo revisitado

Assim como todas as funções no Python, `intersect` é polimórfica – ela funciona em tipos arbitrários, contanto que eles suportem a interface do objeto esperada:

```
>>> x = intersect([1, 2, 3], (1, 4))     # Tipos misturados
>>> x                                    # Objeto resultado salvo
[1]
```

Desta vez, passamos tipos de objetos diferentes para nossa função – uma lista e uma tupla (tipos misturados) – e ela ainda distingue os itens comuns. Como você não precisa especificar os tipos dos argumentos antecipadamente, a função `intersect` faz a iteração em quaisquer tipos de objetos seqüência enviados, desde que eles suportem as interfaces esperadas.

Para `intersect`, isso significa que o primeiro argumento tem de suportar o loop `for` e o segundo tem de suportar o teste de participação como membro `in` – quaisquer dois desses objetos funcionarão. Se você passar objetos que não suportam essas interfaces (passar números, por exemplo), o Python detectará automaticamente a combinação inadequada e lançará uma exceção – exatamente o que você quer e o melhor que poderia fazer, caso desenvolvesse testes de tipo explícitos. A função `intersect` funcionará até em objetos baseados em classe, os quais você vai aprender a construir na Parte VI.*

Variáveis locais

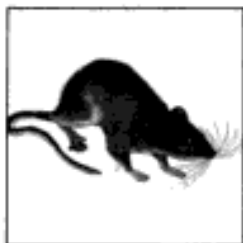
A variável `res` dentro da função `intersect` é o que, no Python, é chamada de *variável local* – um nome visível apenas para o código que está dentro da função `def`, e que só existe enquanto a função é executada. Na verdade, como todos os nomes *atribuídos* de qualquer maneira dentro de uma função são classificados como variáveis locais, por padrão, praticamente todos os nomes em `intersect` são variáveis locais:

* Tecnicamente, a função `intersect` funciona em qualquer objeto que responda ao protocolo de iteração (discutido posteriormente neste capítulo) ou à indexação. O loop `for` e o teste `in` funcionam solicitando um objeto iterador ou indexando um objeto repetidamente. Quando estudarmos as classes, na Parte VI, você vai ver como fazer para implementar esses protocolos para objetos definidos pelo usuário e, assim, suportar `for` e `in`. Quando tiver esse conhecimento, você também poderá passar instâncias de suas classes para a função `intersect`, mesmo para classes que desenvolver no futuro, muito tempo depois que a função tiver sido depurada.

- Como `res` é obviamente atribuída, ela é uma variável local.
- Como os argumentos são passados por atribuição, `seq1` e `seq2` também são.
- Como o loop `for` atribui itens a uma variável, o nome `x` também é.

Todas essas variáveis locais aparecem quando a função é chamada, e desaparecem quando a função termina – instrução `return`, no final de `intersect`, envia de volta o *objeto* resultado, mas o *nome* `res` desaparece. Contudo, para entendermos completamente a noção de variáveis locais, precisamos passar para o Capítulo 13.

13

Escopos e Argumentos

O Capítulo 12 examinou a definição e as chamadas de função básicas. Conforme vimos, o modelo de função básico é simples de usar no Python. Este capítulo apresenta os detalhes existentes por trás dos *escopos* – os lugares onde as variáveis são definidas – do Python, assim como a *passagem de argumentos* – a maneira pela qual os objetos são enviados como entradas para funções.

REGRAS DE ESCOPO

Agora que você vai começar a escrever suas próprias funções, precisamos ser mais formais quanto ao que os nomes significam no Python. Quando você usa um nome em um programa, o Python o cria, altera ou pesquisa no que é conhecido como *espaço de nome* – um lugar onde os nomes ficam. Quando falamos sobre a busca do valor de um nome em relação ao código, o termo *escopo* refere-se a um espaço de nome – a localização da atribuição de um nome em seu código determina o escopo da visibilidade do nome no código.

Praticamente tudo que está relacionado aos nomes acontece em atribuições no Python – até a classificação de escopo. Conforme vimos, os nomes no Python começam a existir quando recebem um valor pela primeira vez, e eles devem ser atribuídos antes de serem usados. Como os nomes não são declarados antecipadamente, o Python utiliza o local da atribuição de um nome para associá-lo (isto é, *vinculá-lo*) a um espaço de nome em particular. Ou seja, o lugar onde você atribui um nome determina o espaço de nome em que ele ficará e, portanto, seu escopo de visibilidade.

Além de empacotar código, as funções adicionam uma camada de espaço de nome extra em seus programas – por padrão, todos os nomes atribuídos dentro de uma função são associados ao espaço de nome dessa função e a nenhum outro. Isso significa que:

- Os nomes definidos dentro de uma instrução `def` só podem ser vistos pelo código que está dentro dessa instrução. Você não pode nem mesmo referir-se a esses nomes fora da função.
- Os nomes definidos dentro de uma instrução `def` não entram em conflito com as variáveis fora dessa instrução, mesmo que um nome igual seja usado em qualquer outro lugar. Um nome `X` atribuído fora de uma instrução `def` é uma variável completamente diferente de um nome `X` atribuído dentro dessa instrução.

O resultado é que os escopos de função ajudam a evitar conflitos de nomes em seus programas e ajudam a transformar as funções em unidades de programa mais auto-suficientes.

Fundamentos de escopo do Python

Antes de você ter começado a escrever funções, todo código era escrito no nível superior de um módulo (isto é, não aninhado em uma instrução `def`), de modo que os nomes ou ficavam no próprio módulo ou eram nomes internos predefinidos pelo Python (por exemplo, `open`).* As funções fornecem um espaço de nome aninhado (isto é, um escopo) que torna locais os nomes que utilizam, de modo que os nomes dentro de uma função não entram em conflito com os que estão fora (em um módulo ou em outra função). As funções definem um escopo local e os módulos definem um escopo global. Os dois escopos são relacionados como segue:

O módulo envolvente é um escopo global. Cada módulo é um escopo global – um espaço de nome onde ficam as variáveis criadas (atribuídas) no nível superior de um arquivo de módulo. As variáveis globais tornam-se atributos de um objeto módulo para o mundo exterior, mas podem ser usadas como variáveis simples dentro de um arquivo.

O escopo global abrange apenas um arquivo. Não se engane com a palavra "global" aqui – os nomes no nível superior de um arquivo são globais apenas para o código que está dentro desse arquivo. Na realidade, no Python não há nenhuma noção de um único escopo global baseado em arquivo abrangendo tudo. Em vez disso, os nomes são particionados em módulos e você sempre deve importar um arquivo explicitamente se quiser usar os nomes que seu arquivo define. Quando você ouvir falar em "global" no Python, pense em "meu módulo".

Cada chamada para uma função é um novo escopo local. Sempre que chama uma função, você cria um novo escopo local – um espaço de nome onde normalmente ficam os nomes criados dentro da função. Você pode considerar isso mais ou menos como se cada instrução `def` (e expressão `lambda`) definisse um novo escopo local. Entretanto, como o Python permite que as funções chamem a si mesmas para fazer um loop – uma técnica avançada, conhecida como recursividade –, tecnicamente o escopo local corresponde a uma chamada de função. Cada chamada cria um novo espaço de nome local. A recursividade é útil ao se processar estruturas cujas formas não podem ser previstas antecipadamente.

Os nomes atribuídos são locais, a não ser que sejam declarados como globais. Por padrão, todos os nomes atribuídos dentro de uma definição de função são colocados no escopo local (o espaço de nome associado à chamada de função). Se você precisar atribuir um nome que fica no nível superior do módulo que envolve a função, poderá fazer isso o declarando em uma instrução global dentro da função.

Todos os outros nomes são locais, globais ou internos envolventes. Os nomes que não recebem um valor na definição da função são considerados como tendo escopo envolvente local (em uma instrução `def` envolvente), global (no espaço de nome do módulo envolvente) ou interno (no módulo de nomes predefinido `__builtin__` predefinido pelo Python).

Note que qualquer tipo de atribuição dentro de uma função classifica um nome como local: instruções `=`, importações, instruções `def`, passagem de argumentos etc. Note também que as alterações no local de objetos não classificam os nomes como locais; somente as atribuições de nome reais classificam. Por exemplo, se o nome `L` é atribuído a uma lista no nível superior

* O código digitado na linha de comando interativa é, na realidade, inserido em um módulo interno chamado `__main__`, de modo que os nomes criados interativamente também ficam em um módulo e, assim, seguem as regras de escopo normais. Há mais informações sobre módulos na Parte V.

de um módulo, uma instrução como `L.append(X)` dentro de uma função não classificará `L` como local, enquanto `L = X` classificará. No primeiro caso, `L` será encontrado no escopo global, como sempre, e alterará a lista global.

Solução de nome: a regra LEGB

Se a seção anterior parece confusa, na verdade ela se resume a três regras simples:

- As referências de nome pesquisam no máximo quatro escopos: local, em seguida as funções envolventes (se houver), depois global e, então, interno.
- As atribuições de nome criam ou alteram nomes locais, por padrão.
- As declarações globais fazem o mapeamento dos nomes atribuídos para o escopo de um módulo envolvente.

Em outras palavras, todos os nomes atribuídos dentro de uma instrução de função `def` (ou `lambda` – uma expressão que conheceremos posteriormente) são locais por padrão; as funções podem usar os nomes em funções *lexicamente* (isto é, fisicamente) envolventes e no escopo global, mas devem declarar como globais para alterá-los. A solução de nome do Python às vezes é chamada de regra LEGB, por causa dos nomes de escopo:

- Quando você usa um nome não qualificado dentro de uma função, o Python pesquisa para cima em quatro escopos – o local (*L*), depois o escopo local de quaisquer instruções `def` e `lambda` envolventes (*E*), em seguida o global (*G*) e, então, o interno (*B*, de built-in) – e pára no primeiro lugar onde o nome é encontrado. Se ele não for encontrado durante essa pesquisa, o Python relatará um erro. Conforme aprendemos no Capítulo 4, os nomes devem ser atribuídos antes de poderem ser utilizados.
- Quando você atribui um nome em uma função (em vez de apenas referir-se a ele em uma expressão), o Python sempre cria ou altera o nome no escopo local, a não ser que ele seja declarado como global nessa função.
- Quando se está fora de uma função (isto é, no nível superior de um módulo ou no prompt interativo), o escopo local é igual ao global – um espaço de nome de módulo.

A Figura 13-1 ilustra os quatro escopos do Python. Note que, tecnicamente, a segunda camada de pesquisa de escopo "E" – instruções `def` e `lambda` envolventes – pode corresponder a mais de uma camada de pesquisa. Ela só entra em ação quando você aninha funções dentro de funções.* Além disso, lembre-se de que essas regras só se aplicam aos nomes de *variável* simples (como `spam`). Nas partes V e VI, veremos que as regras para nomes de *atributo* qualificados (como `objeto.spam`) ficam em um objeto em particular e seguem um conjunto de regras de pesquisa completamente diferente das idéias de escopo abordadas aqui. As referências de atributo (nomes após pontos-finais) pesquisam um ou mais objetos e não escopos, e podem ativar algo chamado herança, discutida na Parte VI.

* A regra de pesquisa de escopo era conhecida como "LGB" na primeira edição deste livro. A camada envolvente `def` foi adicionada posteriormente no Python, para considerar a tarefa de passar explicitamente nomes de escopo envolvente – algo normalmente de interesse marginal para iniciantes no Python.

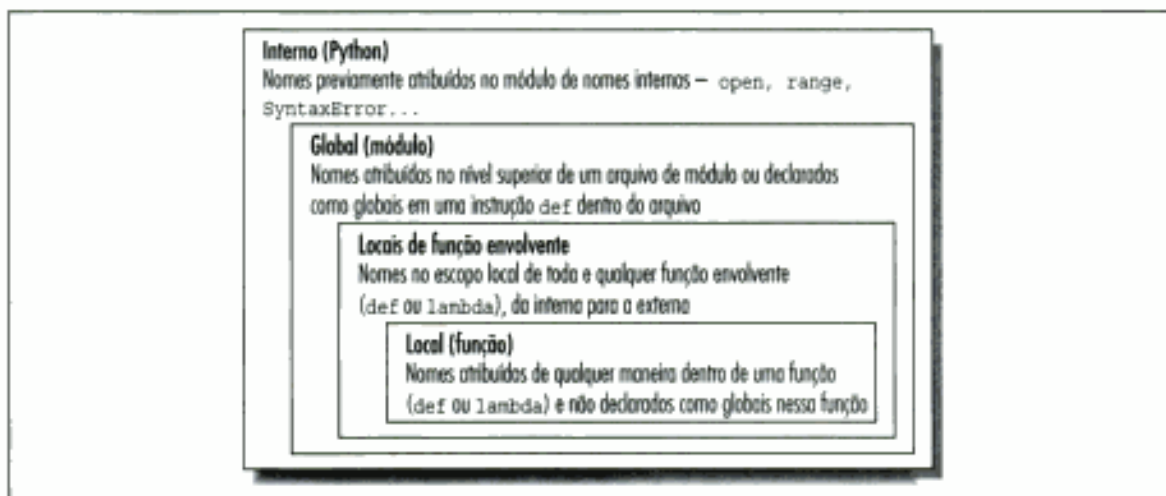



Figura 13-1 A regra de pesquisa de escopo LEGB.

Exemplo de escopo

Vamos ver um exemplo que demonstra as idéias de escopo. Suponha que tenhamos escrito o seguinte código em um arquivo de módulo:

```
# Escopo global
X = 99                          # X e func atribuídos no módulo: globais

def func(Y):                    # Y e Z atribuídos na função: locais
    # escopo local
    Z = X + Y                   # X é global.
    return Z

func(1)                         # func no módulo: resultado=100
```

Esse módulo e a função que ele contém usam vários nomes para fazer seu trabalho. Usando as regras de escopo do Python, podemos classificar os nomes como segue:

Nomes globais: `X`, `func`

`X` é global porque é atribuído no nível superior do arquivo de módulo; ele pode ser referenciado dentro da função sem ser declarado como global. `func` é global pelo mesmo motivo; a instrução `def` atribui um objeto função ao nome `func` no nível superior do módulo.

Nomes locais: `Y`, `Z`

`Y` e `Z` são locais para a função (e só existem enquanto a função é executada), pois recebem um valor na definição da função; `Z`, em virtude da instrução `=`, e `Y` porque os argumentos são sempre passados por atribuição.

A questão principal por trás desse esquema de separação de nomes é que as variáveis locais servem como nomes temporários que você só precisa enquanto a função está em execução. Por exemplo, o argumento `Y` e o resultado da adição `Z` só existem dentro da função; esses nomes não interferem no espaço de nome do módulo envolvente (quanto a isso, nem em qualquer outra função).

A distinção entre local e global também torna uma função mais fácil de entender; a maioria dos nomes que ela usa aparece na própria função e não em algum lugar arbitrário em um módulo. Como você pode ter certeza de que os nomes locais não são alterados por alguma função remota em seu programa, eles também tendem a tornar os programas mais fáceis de depurar.

O escopo interno

Estivemos falando sobre o escopo interno de forma abstrata, mas ele é um pouco mais simples do que você possa imaginar. Na verdade, o escopo interno é apenas um módulo da biblioteca padrão previamente construído, chamado `__builtin__`, que você pode importar e inspecionar se quiser ver quais nomes são predefinidos:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning',
'EOFError', 'Ellipsis',
    ...mais nomes omitidos...
'str', 'super', 'tuple', 'type', 'unichr', 'unicode',
'vars', 'xrange', 'zip']
```

Os nomes dessa lista são o escopo interno do Python; a grosso modo, a primeira metade é composta de exceções internas e a segunda de funções internas. Como o Python pesquisa automaticamente esse módulo por último em sua regra de pesquisa LEGB, você recebe todos os nomes dessa lista gratuitamente – eles podem ser usados sem a importação de nenhum módulo. Na verdade, existem duas maneiras de referir-se a uma função interna: pela regra LEGB ou importando manualmente:

```
>>> zip                            # A maneira normal
<built-in function zip>

>>> import __builtin__             # A maneira difícil
>>> __builtin__.zip
<built-in function zip>
```

A segunda delas às vezes é útil em trabalhos avançados. O leitor cuidadoso também poderá notar que, como o procedimento de pesquisa LEGB pega a primeira ocorrência de um nome que encontra, os nomes no escopo local podem anular variáveis de mesmo nome nos escopos global e interno, e os nomes globais podem anular os internos. Uma função pode, por exemplo, criar uma variável local chamada `open`, atribuindo-a:

```
def hider():
    open = 'spam'                  # Variável local, oculta a interna
    ...
```

Entretanto, isso ocultará a função interna chamada `open` que fica no escopo interno (exterior). Normalmente, esse também é um erro, e grave, pois o Python não emitirá uma mensagem a respeito dele – existem ocasiões, na programação avançada, em que talvez você queira realmente substituir um nome interno, redefinindo-o em seu código.

Analogamente, as funções podem ocultar variáveis globais de mesmo nome, com locais:

```
X = 88                           # X global

def func():
    X = 99                       # X local: oculta o global

func()
print X                          # Imprime 88: inalterado
```

Aqui, a atribuição dentro da função cria um `X` local que é uma variável completamente diferente do `X` global no módulo fora da função. Por isso, não há nenhuma maneira de alterar um nome fora da função, sem adicionar uma declaração global na instrução `def` – conforme descrito na próxima seção.

A INSTRUÇÃO GLOBAL

A instrução global é a única coisa remotamente parecida com uma instrução de declaração no Python. Contudo, não é uma declaração de tipo ou tamanho, mas uma declaração de espaço de nome. Ela informa ao Python que uma função pretende alterar nomes globais – nomes que ficam no escopo (espaço de nome) do módulo envolvente. Já falamos de passagem sobre a instrução global. Como um resumo:

- `global` significa "um nome no nível superior do arquivo de módulo envolvente".
- Os nomes globais só devem ser declarados se forem atribuídos em uma função.
- Os nomes globais podem ser referenciados em uma função sem serem declarados.

A instrução global é apenas a palavra-chave global, seguida por um ou mais nomes, separados por vírgulas. Todos os nomes listados serão mapeados no escopo do módulo envolvente, quando atribuídos ou referenciados dentro do miolo da função. Por exemplo:

```
X = 88                    # X global

def func():
    global X
    X = 99                # X global: fora de def

func()
printX                    # Imprime 99
```

Aqui, adicionamos uma declaração global no exemplo, de modo que o `X` dentro da instrução `def` agora refere-se ao `X` que está fora desta instrução; desta vez, eles são a mesma variável. Aqui está um exemplo, ligeiramente mais complicado, da instrução `global` em funcionamento:

```
y, z = 1, 2                     # Variáveis globais no módulo

def all_global():
    global x                    # Declara globais atribuídas.
    x = y + z                   # Não precisa declarar y,z: regra LEGB
```

Aqui, `s`, `y` e `z` são todas globais dentro da função `all_global.` `y` e `z` são globais porque não são atribuídas na função; `x` é global porque foi listada em uma instrução global para fazer seu mapeamento para o escopo do módulo explicitamente. Sem a instrução global aqui, `x` seria considerada local, graças à atribuição.

Note que `y` e `z` não são declaradas como globais; a regra de pesquisa LEGB do Python as encontra no módulo automaticamente. Note também que `x` poderia não existir no módulo envolvente, antes da execução da função; se não existisse, a atribuição na função criaria `x` no módulo.

Se você quiser alterar nomes fora de funções, terá que escrever código extra (instruções `global`). Por padrão, os nomes atribuídos em funções são locais. Isso é assim por design – como acontece comumente no Python, você precisa escrever mais para fazer a coisa "errada". Embora existam exceções, a alteração de globais pode levar a problemas de engenharia de software bem conhecidos: como os valores das variáveis são dependentes da ordem das chamadas para funções arbitrariamente distantes, os programas podem ser difíceis de depurar. Tente minimizar o uso de globais em seu código.

ESCOPOS E FUNÇÕES ANINHADAS

É hora de examinarmos mais profundamente a letra "E" da regra de pesquisa LEGB. A camada "E" assume a forma dos escopos locais de toda e qualquer instrução `def` de função envol-

vente. Essa camada é um acréscimo relativamente novo no Python (adicionado no Python 2.2) e, às vezes, é denominada de *escopo aninhado estaticamente*. Na realidade, o aninhamento é léxico – escopos aninhados correspondem a estruturas de código fisicamente aninhadas no código-fonte de seu programa.

Detalhes do escopo aninhado

Com a adição de escopos de função aninhados, as regras de pesquisa de variável tornam-se ligeiramente mais complexas. Dentro de uma função:

Atribuição: `X=valor`

Por padrão, cria ou altera o nome `X` no escopo local corrente. Se `X` é declarado global dentro da função, então ela cria ou altera o nome `X` no escopo do módulo envolvente.

Referência: `X`

Procura o nome `X` no escopo (função) local corrente, depois nos escopos locais de todas as funções lexicamente envolventes, da interna para a externa (se houver), em seguida, no escopo global corrente (o arquivo de módulo) e, finalmente, no escopo interno (módulo `__builtin__`). Em vez disso, as declarações `global` fazem a pesquisa começar no escopo global.

Note que a declaração `global` ainda faz o mapeamento de variáveis para o módulo envolvente. Quando funções aninhadas estão presentes, as variáveis nas funções envolventes só podem ser referenciadas e não alteradas. Vamos ilustrar tudo isso com código real.

Exemplos de escopo aninhado

Aqui está um exemplo de escopo aninhado:

```
def f1():
    x = 88
    def f2():
        print x
    f2()

f1()                    # Imprime 88
```

Primeiramente, esse código Python é válido: a instrução `def` é simplesmente uma instrução executável que pode aparecer em qualquer lugar onde qualquer outra instrução puder – inclusive aninhada em outra instrução `def`. Aqui, a instrução `def` aninhada é executada enquanto uma chamada para a função `f1` está em andamento. Ela gera uma função e atribui a ela o nome `f2`, uma variável local dentro do escopo local de `f1`. De certo modo, `f2` é uma função temporária que só existe durante a execução da função `f1` envolvente (e só é visível para o código dessa função).

Mas observe o que acontece dentro de `f2`: quando imprime a variável `x`, ela refere-se à variável `x` que está no escopo local da função `f1` envolvente. Como as funções podem acessar nomes em todas as instruções `def` fisicamente envolventes, a variável `x` em `f2` é automaticamente mapeada na variável `x` que está em `f1`, pela regra de pesquisa LEGB.

Essa pesquisa de escopo envolvente funciona mesmo que a função envolvente já tenha retornado. Por exemplo, o código a seguir define uma função que cria e retorna outra:

```
def f1():
    x = 88
    def f2():
        print x
```

```
    return f2

action = f1()                # Faz e retorna uma função.
action()                     # Agora, a chama: imprime 88
```

Nesse código, a chamada para action está realmente executando a função que chamamos de `f2` quando `f1` foi executada. `f2` lembra de `x` do escopo envolvente em `f1`, mesmo que `f1` não esteja mais ativa. Às vezes, esse tipo de comportamento também é chamado de clausura – um objeto que lembra de valores nos escopos envolventes, mesmo que esses escopos possam não mais existir. Embora as classes (descritas na Parte VI) normalmente sejam melhores para lembrar o estado, tais funções oferecem outra alternativa.

Nas versões anteriores do Python, esse tipo de código falhava, pois as instruções `def` aninhadas não faziam nada a respeito do escopo – uma referência para uma variável dentro de `f2` pesquisaria no escopo local (`f2`), depois no global (o código fora de `f1`) e, então, no interno. Como ela pulava os escopos das funções envolventes, resultava em um erro. Para contornar isso, os programadores normalmente usavam *valores de argumento padrão* para passar (lembrar) os objetos em um escopo envolvente:

```
def f1():
    x = 88
    def f2(x=x):
        print x
    f2()

f1                           # Imprime 88
```

Esse código funciona em todas as versões do Python e você ainda verá esse padrão em muitos códigos Python existentes. Conheceremos os padrões com mais detalhes posteriormente neste capítulo; em resumo, a sintaxe `arg=val` em um cabeçalho `def` significa que o argumento `arg` terá como padrão o valor `val`, caso nenhum valor real seja passado para `arg` em uma chamada.

Na função `f2` modificada, `x=x` significa que o argumento `x` terá como padrão o valor de `x` no escopo envolvente – como o segundo `x` é avaliado antes que o Python percorra a instrução `def` aninhada, ela ainda se refere à variável `x` de `f1`. Na verdade, o padrão se lembra de que `x` estava em `f1`, o objeto `88`.

Isso é bastante complexo e depende totalmente do sincronismo das avaliações de valor padrão. Também é por isso que a regra de pesquisa de escopo aninhado foi adicionada no Python, para tornar os padrões desnecessários para essa tarefa. Atualmente, o Python se lembra automaticamente de todos os valores exigidos no escopo envolvente para uso em instruções `def` aninhadas.

É claro que a melhor prescrição aqui, provavelmente, é simplesmente não fazer isso. Os programas serão muito mais simples se você não aninhar instruções `def` dentro de outras instruções `def`. Aqui está um exemplo equivalente ao anterior, que elimina a noção de aninhamento. Note que é correto chamar uma função definida (veja a seguir) após aquela que contém a chamada, como essa, desde que a segunda instrução `def` seja executada antes da chamada da primeira função – o código dentro de uma instrução `def` nunca é avaliado, até que a função seja realmente chamada:

```
>>> def f1():
...     x = 88
...     f2(x)
...
>>> def f2(x):
...     print x
```

```
...
>>> f1()
88
```

Se você evitar o aninhamento dessa maneira, praticamente poderá se esquecer do conceito dos escopos aninhados do Python, pelo menos para instruções `def`. Entretanto, é ainda mais provável que você se preocupe com essas coisas quando começar a escrever expressões `lambda`. Também não conheceremos a expressão `lambda` em profundidade até o Capítulo 14. Em resumo, `lambda` é uma expressão que gera uma nova função para ser chamada posteriormente, de forma muito parecida com uma instrução `def` (como se trata de uma expressão, ela pode ser usada em lugares que a instrução `def` não pode, como dentro de literais de lista e dicionário).

Também como uma instrução `def`, as expressões `lambda` introduzem um novo escopo local. Com a camada de pesquisa de escopos envolventes, elas podem ver todas as variáveis que ficam na função em que são escritas. O que segue só funciona hoje porque as regras de escopo aninhado são aplicadas agora:

```
def func():
    x = 4
    action = (lambda n: x ** n)          # x na instrução def envolvente
    return action

x = func()
print x(2)                               # Imprime 16
```

Antes da introdução dos escopos de função aninhados, os programadores usavam padrões para passar valores de um escopo envolvente para expressões `lambda`, assim como para instruções `def`. Por exemplo, o que segue funciona em todas as versões do Python:

```
def func()
    x = 4
    action = (lambda n, x=x: x ** n)     # Passa x manualmente.
```

Como as `lambda` são expressões, naturalmente (e mesmo normalmente) elas são aninhadas dentro de instruções `def` envolventes. Assim, talvez elas sejam as maiores beneficiárias da inclusão dos escopos de função envolventes nas regras de pesquisa. Na maioria dos casos, não é mais necessário passar valores com padrões para expressões `lambda`. Teremos mais a dizer sobre padrões e sobre expressões `lambda` posteriormente, de modo que talvez você queira retornar e rever esta seção depois.

Antes de terminarmos esta discussão, note que os escopos são aninhados arbitrariamente, mas apenas as funções envolventes (e não classes, descritas na Parte VI) são pesquisadas:

```
>>> def f1():
...     x = 99
...     def f2():
...         def f3():
...             print x         # Encontrado no escopo local de f1!
...         f3()
...     f2()
...
>>> f1()
99
```

O Python pesquisará os escopos locais de *todas* as instruções `def` envolventes, após o escopo local da função que está fazendo a referência e antes do escopo global do módulo. Entretanto, esse tipo de código parece menos provável na prática.

PASSANDO ARGUMENTOS

Vamos expandir a noção de passagem de argumentos no Python. Anteriormente, observamos que os argumentos são passados por atribuição. Isso tem algumas ramificações que nem sempre são evidentes para os iniciantes:

Os argumentos são passados pela atribuição automática de objetos a nomes locais. Os argumentos de função são apenas outro caso da atribuição do Python em funcionamento. Os argumentos de função são referências para objetos (possivelmente) compartilhados, referenciados por quem fez a chamada.

A atribuição de nomes de argumento dentro de uma função não afeta quem fez a chamada. Os nomes de argumento no cabeçalho da função tornam-se novos nomes locais quando a função é executada no escopo da função. Não há nenhum alias entre os nomes de argumento da função e os nomes que estão em quem fez a chamada.

A alteração de um argumento de objeto mutável em uma função pode ter impacto sobre quem fez a chamada. Por outro lado, como os argumentos são simplesmente atribuídos a objetos passados, as funções podem alterar objetos mutáveis passados e o resultado pode afetar quem fez a chamada.

O esquema de passagem por atribuição do Python não é igual aos parâmetros de referência da linguagem C++, mas verifica-se que é muito semelhante aos argumentos da linguagem C, na prática:

Os argumentos imutáveis agem como o modo "por valor" da linguagem C. Objetos como inteiros e strings são passados por referência (atribuição) de objeto, mas como, de qualquer maneira, você não pode alterar objetos imutáveis no local, o efeito é muito parecido com fazer uma cópia.

Os argumentos mutáveis agem como o modo "por ponteiro" da linguagem C. Objetos como listas e dicionários são passados por referência de objeto, que é semelhante ao modo como a linguagem C passa arrays como ponteiros – os objetos mutáveis podem ser alterados no local na função, de forma muito parecida com os arrays da linguagem C.

Naturalmente, se você nunca usou a linguagem C, o modo de passagem de argumentos do Python será ainda mais simples – trata-se apenas de uma atribuição de objetos a nomes, que funciona da mesma forma, sejam os objetos mutáveis ou não.

Argumentos e referências compartilhadas

Aqui está um exemplo que ilustra algumas dessas propriedades em funcionamento:

```
>>> def changer(x, y):          # Função
...     x = 2                   # Altera apenas o valor do nome local
...     y[0] = 'spam'           # Altera o objeto compartilhado no local
...
>>> X = 1
>>> L = [1, 2]                  # Quem faz a chamada
>>> changer(X, L)               # Passa imutável e mutável
>>> X, L                        # X inalterado, L é diferente
(1, ['spam', 2])
```

Nesse código, a função `changer` atribui o nome de argumento `x` e um componente no objeto referenciado pelo argumento `y`. As duas atribuições dentro da função têm sintaxe apenas ligeiramente diferentes, mas seus resultados são radicalmente distintos:

- Como `x` é um nome local no escopo da função, a primeira atribuição não tem efeito sobre quem fez a chamada – ela simplesmente altera a variável local `x` e não altera o vínculo do nome `X` em quem fez a chamada.
- O argumento `y` também é um nome local, mas é passado um objeto mutável (a lista chamada `L` em quem fez a chamada). Como a segunda atribuição é uma alteração de objeto no local, o resultado da atribuição para `y[0]` na função tem impacto no valor de `L`, depois que a função retorna.

A Figura 13-2 ilustra os vínculos nome/objeto que existem imediatamente após a função ter sido chamada e antes que seu código seja executado.

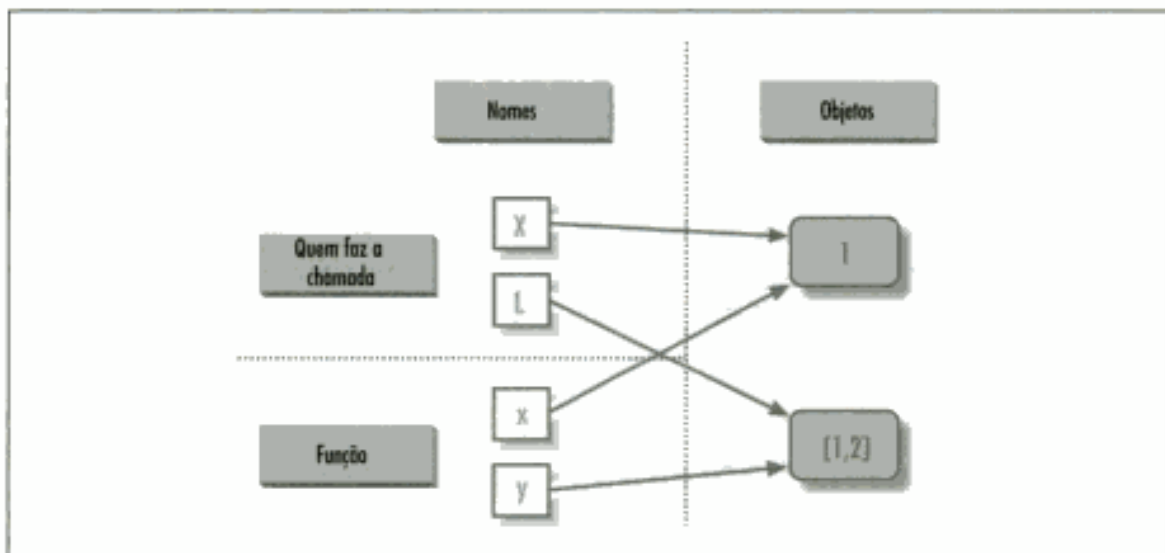



Figura 13-2 Referências: os argumentos compartilham objetos com quem fez a chamada.

Se você se lembrar da discussão sobre objetos mutáveis compartilhados nos capítulos 4 e 7, reconhecerá que esse é exatamente o mesmo fenômeno ocorrendo: alterar um objeto mutável no local pode afetar outras referências para o objeto. Aqui, seu efeito é fazer um dos argumentos funcionar como uma *saída* da função.

Evitando alterações de argumento mutável

Se você não quer que as alterações no local dentro de funções afetem os objetos passados para elas, basta fazer cópias explícitas dos objetos mutáveis, conforme aprendemos no Capítulo 7. Para argumentos de função, podemos copiar a lista no ponto da chamada:

```
L = [1, 2]
changer(X, L[:])            # Passa uma cópia; portanto, meu L não muda.
```

Também podemos copiar dentro da própria função, se nunca quisermos alterar os objetos passados, independente de como a função seja chamada:

```
def changer(x, y):
   y = y[:]                 # Copia a lista de entrada; portanto, não afeta
                            quem fez a chamada.
   x = 2
   y[0] = 'spam'            # Altera apenas a cópia de minha lista
```

Esses dois esquemas de cópia não impedem a função de alterar o objeto – eles apenas impedem que essas alterações afetem quem fez a chamada. Para realmente evitar alterações,

sempre podemos converter para objetos imutáveis, para forçar o resultado. As tuplas, por exemplo, lançam uma exceção quando tentamos fazer alterações:

```
L = [1, 2]
changer(X, tuple(L))     # Passa uma tupla; portanto as alterações são erros.
```

Esse esquema usa a função interna `tuple`, que constrói uma nova tupla a partir de todos os itens de uma seqüência. Isso também é algo extremo – como força a função a ser gravada a nunca alterar os argumentos passados, isso pode impor mais limitações sobre a função do que deveria e, freqüentemente, deve ser evitado. Você nunca sabe quando alterar argumentos poderia ser útil para outras chamadas no futuro. A função também perderá a capacidade de chamar quaisquer métodos específicos de lista no argumento, mesmo métodos que não alteram o objeto no local.

Simulando parâmetros de saída

Já discutimos a instrução `return` e a utilizamos em alguns exemplos, mas aqui está um truque: como a instrução `return` envia de volta qualquer tipo de objeto, ela pode retornar vários valores, se eles forem empacotados em uma tupla. De fato, embora o Python não tenha o que algumas linguagens chamam de passagem de argumento com chamada por referência, normalmente podemos simulá-la retornando tuplas e atribuindo os resultados de volta aos nomes de argumento originais em quem fez a chamada:

```
>>> def multiple(x, y):
...     x = 2                   # Altera apenas nomes locais
...     y = [3, 4]
...     return x, y             # Retorna novos valores em uma tupla.
...
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)       # Atribui os resultados aos nome de
                                quem fez a chamada.
>>> X, L
(2, [3, 4])
```

Aqui, parece que o código está retornando dois valores, mas é apenas um – uma tupla de dois itens, com os parênteses circundantes opcionais omitidos. Depois que a chamada retornar, use atribuição de tupla para desempacotar as partes da tupla retornada. (Se você tiver esquecido porque, volte para as seções "Tuplas" no Capítulo 7 e "Instruções de atribuição" no Capítulo 8.) O resultado desse padrão de codificação é a simulação dos parâmetros de saída de outras linguagens por meio de atribuições explícitas. `X` e `L` mudam após a chamada, mas somente porque o código disse para fazer isso.

MODOS ESPECIAIS DE CORRESPONDÊNCIA DE ARGUMENTO

Os argumentos são sempre passados por *atribuição* no Python; os nomes no cabeçalho de `def` são atribuídos aos objetos passados. Em cima desse modelo, contudo, o Python fornece ferramentas adicionais que alteram a maneira como é feita a *correspondência* dos objetos argumento da chamada com os nomes de argumento no cabeçalho, antes da atribuição. Todas essas ferramentas são opcionais, mas permitem escrever funções que suportam padrões de chamada mais flexíveis.

Por padrão, a correspondência dos argumentos é feita pela posição, da esquerda para a direita, e você deve passar exatamente o mesmo número de argumentos quantos forem os nomes de argumento no cabeçalho da função. Mas você também pode especificar uma correspondência pelo nome, valores padrão e coletores para argumentos extras.

Parte desta seção é complicada e, antes de entrarmos nos detalhes sintáticos, gostaríamos de enfatizar que esses modos especiais são opcionais e estão relacionados apenas com a correspondência de objetos com nomes. O mecanismo de passagem subjacente ainda é a atribuição, após a correspondência ocorrer. Mas aqui está uma sinopse dos modos de correspondência disponíveis:

Posicionais: correspondência da esquerda para a direita
: O caso normal, usado até aqui, é fazer a correspondência dos argumentos pela posição.

Palavras-chave: correspondência pelo nome do argumento
: Quem faz a chamada pode especificar qual argumento na função deve receber um valor, usando o nome do argumento na chamada, com a sintaxe `nome=valor`.

Varargs: captura argumentos posicionais ou de palavra-chave que não correspondem
: As funções podem usar argumentos especiais precedidos por caracteres `*`, para coletar arbitrariamente argumentos extras (muito parecido – e freqüentemente denominado como – com o recurso *varargs* da linguagem C, que suporta listas de argumentos de comprimento variável).

Padrões: especifica valores para argumentos que não são passados
: As funções também podem especificar valores padrão de argumentos a receber, caso a chamada passe valores insuficientes, usando a sintaxe `nome=valor`.

A Tabela 13-1 resume a sintaxe que ativa os modos de correspondência especiais.

Tabela 13-1 Formas de correspondência de argumentos de função

Sintaxe	Localização	Interpretação
`func(value)`	Quem faz a chamada	Argumento normal: correspondência pela posição
`func(nome=valor)`	Quem faz a chamada	Argumento de palavra-chave: correspondência pelo nome
`def func(nome)`	Função	Argumento normal: correspondência pela posição ou pelo nome
`def func(nome=valor)`	Função	Valor de argumento padrão, se não for passado na chamada
`def func(*nome)`	Função	Corresponde aos args posicionais restantes (em uma tupla)
`def func (**nome)`	Função	Corresponde aos args de palavra-chave restantes (em um dicionário)

Em uma chamada (as duas primeiras linhas da tabela), a correspondência de nomes simples é feita pela posição, mas usar a forma `nome=valor` diz ao Python que faça a correspondência pelo nome. Esses são chamados de *argumentos de palavra-chave*.

Em um cabeçalho de função, a correspondência de nomes simples é feita pela posição ou pelo nome (dependendo de como quem faz a chamada os passa), mas a forma `nome=valor` especifica um valor padrão, `*nome` coleta todos os argumentos posicionais extras em uma tupla e a forma `**nome` coleta os argumentos de palavra-chave extras em um dicionário.

Como resultado, os modos de correspondência especiais permitem que você seja bastante liberal com relação a quantos argumentos devem ser passados para uma função. Se uma função especifica padrões, eles serão usados se você passar argumentos insuficientes. Se uma função usa a forma *varargs* (lista de argumentos variável), você pode passar argumentos de sobra; os nomes varargs coletam os argumentos extras em uma estrutura de dados.

Exemplos de palavra-chave e padrão

Por padrão, o Python faz a correspondência de nomes pela posição, assim como a maioria das outras linguagens. Por exemplo, se você definir uma função que exige três argumentos, então deve chamá-la com três argumentos:

```
>>> def f(a, b, c): print a, b, c
...
```

Aqui, os passamos pela posição: a corresponde a 1, b corresponde a 2 e etc.:

```
>>> f(1, 2, 3)
1 2 3
```

Contudo, no Python você pode ser mais específico quanto ao que vai onde, quando chama uma função. Os argumentos de palavra-chave nos permitem fazer a correspondência pelo *nome*, em vez da posição:

```
>>> f(c=3, b=2, a=1)
1 2 3
```

Aqui, c=3 na chamada significa a correspondência disso com o argumento chamado c no cabeçalho. O resultado dessa chamada é o mesmo de antes, mas note que a ordem da esquerda para a direita não importa mais quando são usadas palavras-chave, pois a correspondência dos argumentos é pelo nome e não pela posição. É até possível combinar argumentos posicionais e de palavra-chave em uma única chamada – a correspondência de todos os argumentos posicionais é feita primeiro, da esquerda para a direita no cabeçalho, antes da correspondência dos argumentos de palavra-chave pelo nome:

```
>>> f(1, c=3, b=2)
1 2 3
```

Quando vê isso pela primeira vez, a maioria das pessoas se pergunta por que alguém usaria essa ferramenta. Normalmente, as palavras-chave têm duas funções no Python. Primeiramente, elas tornam suas chamadas um pouco mais auto-documentadas (supondo que você utilize nomes melhores do que a, b e c). Por exemplo, uma chamada desta forma:

```
func(name='Bob', age=40, job='dev')
```

é muito mais significativa do que uma chamada com três valores sem nada, separados por vírgulas – as palavras-chave servem como rótulos para os dados na chamada. O segundo principal uso das palavras-chave ocorre em conjunto com padrões.

Apresentamos os padrões anteriormente, quando discutimos os escopos de função aninhados. Em resumo, os padrões nos permitem tornar opcionais os argumentos de função selecionados; se não for passado um valor, o argumento recebe seu padrão antes da execução da função. Por exemplo, aqui está uma função que exige um argumento e dois recebem o padrão:

```
>>> def f(a, b=2, c=3): print a, b, c
...
```

Quando chamada, devemos fornecer um valor para a, ou pela posição ou palavra-chave. Se não passarmos valores para b e c, eles terão 2 e 3 como padrão, respectivamente:

```
>>> f(1)
1 2 3
>>> f(a=1)
1 2 3
```

Se passarmos dois valores, apenas c receberá seu padrão. Com três valores, nenhum padrão será usado:

```
>>> f(1, 4)
1 4 3
>>> f(1, 4, 5)
1 4 5
```

Finalmente, aqui está como os recursos de palavra-chave e padrão interagem: como subvertem o mapeamento posicional da esquerda para a direita normal, as palavras-chave nos permitem basicamente pular sobre argumentos com padrões:

```
>>> f(1, c=6)
1 2 6
```

Aqui, a recebe 1 pela posição, c recebe 6 pela palavra-chave e b, entre eles, tem 2 como padrão.

Exemplos de argumentos arbitrários

As duas últimas extensões de correspondência, * e **, são projetadas para suportar funções que recebem qualquer número de argumentos. A primeira extensão coleta em uma tupla os argumentos posicionais sem correspondência:

```
>>> def f(*args): print args
```

Quando essa função é chamada, o Python coleta todos os argumentos posicionais em uma nova tupla e atribui a variável args a essa tupla. Como se trata de um objeto tupla normal, ele poderia ser indexado, percorrido com um loop for etc.:

```
>>> f()
()
>>> f(1)
(1,)
>>> f(1,2,3,4)
(1, 2, 3, 4)
```

O recurso ** é semelhante, mas só funciona para argumentos de palavra-chave – ele os coleta em um novo dicionário, o qual pode então ser processado com as ferramentas de dicionário normais. De certo modo, a forma ** permite converter de palavras-chave para dicionários:

```
>>> def f(**args): print args
...
>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Finalmente, os cabeçalhos de função podem combinar argumentos normais, * e ** para implementar assinaturas de chamada extremamente flexíveis:

```
>>> def f(a, *pargs, **kargs): print a, pargs, kargs
...
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

Na verdade, esses recursos podem ser combinados de maneiras mais complexas que, à primeira vista, parecem quase ambíguas – uma idéia que reveremos posteriormente neste capítulo.

Combinando palavras-chave e padrões

Aqui está um exemplo ligeiramente maior que demonstra palavras-chave e padrões em ação. No código a seguir, quem faz a chamada sempre deve passar pelo menos dois argumentos (para corresponder a spam e eggs), mas os outros dois são opcionais; se forem omitidos, o Python atribuirá os padrões especificados no cabeçalho a toast e a ham:

```
def func(spam, eggs, toast=0, ham=0):      # Os 2 primeiros são obrigatórios
    print (spam, eggs, toast, ham)

func(1, 2)                                 # Saída: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                     # Saída: (1, 0, 0, 1)
func(spam=1, eggs=0)                       # Saída: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)              # Saída: (3, 2, 1, 0)
func(1, 2, 3, 4)                           # Saída: (1, 2, 3, 4)
```

Note que, quando são usados argumentos de palavra-chave na chamada, a ordem na qual eles são listados não importa. O Python faz a correspondência pelo nome e não pela posição. Quem faz a chamada deve fornecer valores para `spam` e `eggs`, mas sua correspondência pode ser pela posição ou pelo nome. Note também que a forma `nome=valor` significa coisas diferentes na chamada e que `def`: é uma palavra-chave na chamada e um padrão no cabeçalho.

A chamada para despertar max

Para tornarmos isso mais concreto, vamos trabalhar em um exercício que demonstra uma aplicação prática das ferramentas de correspondência de argumento. Suponha que você queira codificar uma função que seja capaz de calcular o valor mínimo de um conjunto de argumentos arbitrário e um conjunto arbitrário de tipos de dados de objeto. Isto é, a função deve aceitar zero ou mais argumentos – tantos quantos você quiser passar. Além disso, a função deve funcionar para todos os tipos de objeto do Python – números, strings, listas, listas de dicionários, arquivos e até `None`.

A primeira parte disso fornece um exemplo natural de como o recurso `*` pode ser utilizado – podemos coletar argumentos em uma tupla e percorrer cada um por sua vez, com um loop `for` simples. A segunda parte da definição desse problema é fácil: como todo tipo de objeto suporta comparações, não precisamos especializar a função por tipo (uma aplicação do *polimorfismo*); basta comparar os objetos cegamente e deixar o Python executar a ordenação de comparação correta.

Crédito completo

O arquivo a seguir mostra três maneiras de codificar essa operação, pelo menos uma das quais foi sugerida por um aluno em algum momento:

- Na primeira função, pegue o primeiro argumento (`args` é uma tupla) e percorra o restante fracionando o primeiro (não há porque comparar um objeto com ele mesmo, especialmente se ele pode ser uma estrutura grande).
- A segunda versão permite que o Python extraia o primeiro e o restante dos argumentos automaticamente e, portanto, evita um índice e um fracionamento.
- A terceira converte de tupla para lista com a chamada interna `list` e emprega o método de lista `sort`.

Como o método `sort` é muito rápido, este terceiro esquema normalmente será o mais veloz, quando tiver sido cronometrado. O arquivo *mins.py* contém o código da solução:

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res
```

```
def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first

def min3(*args):
    tmp = list(args)
    tmp.sort()
    return tmp[0]

print min1(3,4,1,2)
print min2("bb", "aa")
print min3([2,2], [1,1], [3,3])
```

Todas as três produzem o mesmo resultado quando executamos esse arquivo. Tente digitar algumas chamadas interativamente para experimentá-las por conta própria.

```
C:\Python22>python mins.py
1
aa
[1, 1]
```

Note que nenhuma dessas três variantes testa o caso em que nenhum argumento é passado. Elas poderiam fazer isso, mas não há porque fazer isso aqui – em todas as três, o Python lançará uma exceção automaticamente, caso nenhum argumento seja passado. A primeira lança uma exceção quando tentamos buscar o item 0; a segunda, quando o Python detecta um descasamento na lista de argumentos; e a terceira, quando tentamos retornar o item 0 no final.

Contudo, é exatamente isso que queremos que aconteça – como essas funções suportam qualquer tipo de dados, não existe nenhum valor de sentinela válido que pudéssemos passar de volta para designar um erro. Há exceções a essa regra (por exemplo, se você tiver que executar ações dispendiosas antes de chegar ao erro), mas, em geral, é melhor supor que os argumentos funcionarão no código de suas funções e deixar o Python gerar erros para você quando não funcionarem.

Pontos de bônus

Os alunos e leitores podem receber pontos de bônus aqui, alterando essas funções para calcular os valores *máximos*, em vez dos mínimos. Ora, isso é muito fácil: as duas primeiras versões só exigem alterar `< para >` e a terceira só exige que retornemos `tmp[-1]`, em vez de `tmp[0]`. Para ganhar pontos extras, configure também o nome da função para "`max`" (embora essa parte seja estritamente opcional).

É possível generalizar uma única função como essa, para calcular um mínimo ou um máximo, avaliando strings de expressão de comparação com ferramentas como a função interna `eval` ou passando uma função de comparação arbitrária. O arquivo *minmax.py* mostra como se faz para implementar este último esquema:

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y        # veja também: lambda
def grtrthan(x, y): return x > y
```

```
print minmax(lessthan, 4, 2, 1, 5, 6, 3)
print minmax(grtrthan, 4, 2, 1, 5, 6, 3)

% python minmax.py
1
6
```

As funções são outro tipo de objeto que pode ser passado para outra função como essa. Para torná-la uma função `max` (ou outra), simplesmente passamos o tipo correto de função `test`.

A linha perfurada

É claro que este é apenas um exercício de codificação. Na verdade, não há nenhuma razão para codificar funções `min` ou `max`, pois ambas são internas no Python! As versões internas funcionam quase exatamente como as suas, mas são codificadas em C para obter uma velocidade ótima.

Um exemplo mais útil: funções de conjunto gerais

Aqui está um exemplo mais útil dos modos de correspondência de argumentos especiais em funcionamento. Anteriormente, neste capítulo, escrevemos uma função que retornava a interseção de duas seqüências (ela extraía os itens que apareciam nas duas). Aqui está uma versão que faz a interseção de um número de seqüências arbitrário (uma ou mais), usando a forma de correspondência varargs `*args` para coletar todos os argumentos passados. Como os argumentos vêm como uma tupla, podemos processá-los em um loop `for` simples. Apenas por brincadeira, também codificamos uma função de união para um número arbitrário de argumentos. Ela coleta os itens que aparecem em qualquer um dos operandos:

```
def intersect(*args):
    res = []
    for x in args[0]:                   # Percorre a primeira seqüência
        for other in arg [1:]:          # para todos os outros args.
            if x not in other: break    # Item em cada uma?
        else:                           # Não: sai do loop
            res.append(x)               # Sim: adiciona items no fim
    return res

def union(*args):
    res = []
    for seq in args:                    # Para todos os args
        for x in seq:                   # Para todos os nós
            if not x in res:
                res.append(x)           # Adiciona novos itens no resultado.
    return res
```

Como essas são ferramentas que vale a pena reutilizar (e são grandes demais para digitar outra vez interativamente), armazenamos as funções em um arquivo de módulo chamado *inter2.py* (mais informações sobre módulos aparecem na Parte V). Nas duas funções, os argumentos passados na chamada vêm como a tupla `args`. Assim como na interseção original, ambas funcionam em qualquer tipo de seqüência. Aqui, elas estão processando strings, tipos misturados e mais de duas seqüências:

```
% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"

>>> intersect(s1, s2), union(s1, s2)            # Dois operandos
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])
```

```
>>> intersect([1,2,3], (1,4))                    # Tipos misturados
[1]

>>> intersect(s1, s2, s3)                        # Três operandos
['S', 'A', 'M']

>>> union(s1, s2, s3)
['S', 'P', 'A', 'M', 'C', 'L']
```

Correspondência de argumentos: os detalhes arenosos

Se você optar por usar e combinar os modos de correspondência de argumentos especiais, o Python pedirá para que siga duas regras de ordenação:

- Em uma chamada, todos os argumentos de palavra-chave devem aparecer depois de todos os argumentos que não sejam de palavra-chave.
- Em um cabeçalho de função, `*nome` deve aparecer após os argumentos e padrões normais e `*nome` deve aparecer por último.

Além disso, internamente, o Python executa os seguintes passos para fazer a correspondência de argumentos, antes da atribuição:

1. Atribui pela posição os argumentos que não são de palavra-chave.
2. Atribui pela correspondência de nomes os argumentos de palavra-chave.
3. Atribui os argumentos extras que não são de palavra-chave à tupla `*nome`.
4. Atribui os argumentos de palavra-chave extras ao dicionário `**nome`.
5. Atribui valores padrão aos argumentos não atribuídos no cabeçalho.

Isso é tão complicado quanto parece, mas acompanhar o algoritmo de correspondência do Python ajuda a entender alguns casos, especialmente quando os modos são misturados. Vamos deixar os exemplos adicionais desses modos de correspondência especiais para quando fizermos os exercícios no final da Parte IV.

Conforme você pode ver, os modos de correspondência de argumentos avançados podem ser complexos. Eles também são inteiramente opcionais; você pode arranjar-se apenas com a correspondência posicional simples e provavelmente é uma boa idéia fazer isso, caso esteja apenas começando. Entretanto, como algumas ferramentas do Python os utilizam, é importante conhecê-los em geral.

Por que isto é relevante: argumentos de palavra-chave

Os argumentos de palavra-chave desempenham um papel importante na Tkinter, a API de GUI padrão de fato do Python. Conheceremos a Tkinter posteriormente neste livro, mas, como uma prévia, os argumentos de palavra-chave definem opções de configuração quando os componentes da GUI são construídos. Por exemplo, uma chamada da forma:

```
from Tkinter import *
widget = Button(text="Press me", command=someFunction)
```

cria um novo botão e especifica seu texto e sua função de callback, usando os argumentos de palavra-chave `text` e `command`. Como o número de opções de configuração para um elemento de janela pode ser grande, os argumentos de palavra-chave permitem que você selecione e escolha. Sem eles, talvez você tivesse que listar todas as opções possíveis pela posição ou esperar por um protocolo de padrões de argumento posicional criterioso que tratasse de cada arranjo de opção possível.

14 Tópicos de Função Avançados

Este capítulo apresenta uma coleção de tópicos mais avançados relacionados às funções: a expressão `lambda`, ferramentas de programação funcional, como mapas e abrangências de lista, funções geradoras e muito mais. Parte da arte de usar funções reside nas interfaces existentes entre elas; portanto, também exploraremos aqui alguns princípios gerais de projeto de função. Como este é o último capítulo da Parte IV, encerraremos com os conjuntos normais de problemas e exercícios para ajudá-lo a começar a desenvolver as idéias sobre as quais leu.

FUNÇÕES ANÔNIMAS: LAMBDA

Até aqui, vimos o que é necessário para escrever nossas próprias funções em Python. Nas próximas seções, vamos ver algumas idéias mais avançadas relacionadas às funções. São recursos opcionais, em sua maior parte, mas podem simplificar suas tarefas de desenvolvimento, quando bem usados.

Além da instrução `def`, o Python também fornece uma forma de *expressão* que gera objetos função. Por causa de sua semelhança com uma ferramenta da linguagem LISP, ela é chamada de `lambda`.* Assim como a instrução `def`, essa expressão cria uma função para ser chamada posteriormente, mas a retorna, em vez de atribuir um nome a ela. É por isso que, às vezes, as expressões `lambda` são conhecidas como funções anônimas (isto é, sem nome). Na prática, elas são freqüentemente usadas como uma maneira de colocar uma definição de função em linha ou adiar a execução de um trecho de código.

Expressões lambda

A forma geral de `lambda` é a palavra-chave `lambda`, seguida de um ou mais argumentos (exatamente como a lista de argumentos que você coloca entre parênteses em um cabeçalho de `def`), seguido(s) de uma expressão após os dois-pontos:

```
lambda argumento1, argumento2,... argumento N: expressão usando argumentos
```

* O nome "lambda" parece assustar as pessoas mais do que deveria. Ele vem da linguagem LISP, que o trouxe do cálculo de `lambda`, uma forma de lógica simbólica. No Python, contudo, trata-se na realidade apenas de uma palavra-chave que introduz a expressão sintaticamente.

Os objetos função retornados pela execução de expressões `lambda` funcionam de maneira exatamente igual àqueles criados e atribuídos pela instrução `def`. Mas a expressão `lambda` tem algumas diferenças que a tornam útil em tarefas especializadas:

`lambda` *é uma expressão e não uma instrução.* Por isso, uma expressão `lambda` pode aparecer em lugares em que uma instrução `def` não é permitida pela sintaxe do Python – dentro de uma literal de lista ou chamada de função, por exemplo. Como uma expressão, `lambda` retorna um valor (uma nova função), ao qual, opcionalmente, pode ser atribuído um nome. A instrução `def` sempre atribui à nova função o nome que está no cabeçalho, em vez de retorná-la como resultado.

O miolo de* `lambda` *é uma única expressão e não um bloco de instruções. O miolo de `lambda` é semelhante ao que você colocaria na instrução `return` do miolo de uma instrução `def`. Basta digitar o resultado como uma expressão simples, em vez de de retorná-lo explicitamente. Como é limitada a uma expressão, `lambda` é menos geral do que uma instrução `def`. Você só pode colocar lógica no miolo de uma expressão `lambda` usando instruções como `if` (continue a ler para obter mais informações sobre isso). Isso é assim por design, para limitar o aninhamento do programa: `lambda` é projetada para desenvolver funções simples e `def` trata de tarefas maiores.

Fora essas distinções, `def` e `lambda` fazem o mesmo tipo de trabalho. Por exemplo, vimos como fazer funções com instruções `def`:

```
>>> def func(x, y, z): return x + y + z
...
>>> func(2, 3, 4)
9
```

Mas você pode obter o mesmo efeito com uma expressão `lambda`, atribuindo seu resultado explicitamente a um nome, por meio do qual você chamar posteriormente:

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

Aqui, `f` recebe o objeto função criado pela expressão `lambda`; é assim que a instrução `def` funciona também, mas sua atribuição é automática. Os padrões funcionam nos argumentos de `lambda`, exatamente como na instrução `def`:

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefiefoe'
```

O código no miolo de uma expressão `lambda` também segue as mesmas regras de pesquisa de escopo do código dentro de uma instrução `def` – as expressões `lambda` introduzem um escopo local exatamente como uma instrução `def` aninhada, que vê automaticamente nomes no escopo das funções envolventes, do módulo e interno (por meio da regra LEGB):

```
>>> def knights():
...     title = 'Sir'
...     action = (lambda x: title + ' ' + x)      # Title na instrução
                                                    def envolvente
...     return action                             # Retorna uma função.
...
>>> act = knights()
>>> act('robin')
'Sir robin'
```

Antes da versão 2.2, em vez disso, o valor do nome `title` normalmente seria passado como um valor de argumento padrão. Volte ao estudo dos escopos do Capítulo 13, se você se esqueceu o motivo disso.

Por que lambda?

Geralmente falando, as expressões `lambda` são úteis como uma espécie de atalho de função que permite incorporar a definição de uma função dentro do código que a utiliza. Elas são inteiramente opcionais (você sempre pode usar a instrução `def`, em vez disso), mas tendem a ser uma construção de codificação mais simples nos cenários em que você precisa apenas incorporar pequenos trechos de código executável.

Por exemplo, veremos posteriormente que rotinas de tratamento de callback são freqüentemente escritas como expressões `lambda` em linha, incorporadas diretamente na lista de argumentos da chamada de registro, em vez de serem definidas com uma instrução `def` em algum lugar de um arquivo e referenciadas pelo nome (consulte o quadro sobre callbacks para ver um exemplo).

As expressões `lambda` também são comumente usadas para desenvolver *tabelas de salto* – listas ou dicionários de ações a serem executadas de acordo com a demanda. Por exemplo:

```
L = [(lambda x: x**2), (lambda x: x**3), (lambda x: x**4)]

for f in L:
    print f(2)          # Imprime 4, 8, 16

print L[0](3)           # Imprime 9
```

A expressão `lambda` é mais útil como um atalho para `def`, quando você precisa colocar pequenos trechos de código executável em lugares onde as instruções são sintaticamente inválidas. Esse trecho de código, por exemplo, constrói uma lista de três funções, incorporando expressões `lambda` dentro de uma literal de lista; a instrução `def` não funcionaria dentro de uma literal de lista como essa, pois é uma instrução e não uma expressão.

Você pode fazer o mesmo tipo de coisa para construir tabelas de ações, com dicionários e outras estruturas de dados no Python:

```
>>> key = 'got'
>>> {'already':  (lambda: 2 + 2),
...  'got':      (lambda: 2 * 4),
...  'one':      (lambda: 2 ** 6)
... }[key]()
8
```

Aqui, quando o Python faz o dicionário, cada uma das expressões aninhadas `lambda` gera e deixa para trás uma função a ser chamada posteriormente. A indexação pela chave busca uma dessas funções e os parênteses obrigam a função buscada a ser chamada. Quando desenvolvemos dessa maneira, um dicionário torna-se uma ferramenta de *desvio de vários caminhos* mais geral do que a que pudemos mostrar no estudo sobre as instruções `if`, no Capítulo 9.

Para fazer isso funcionar sem `lambda`, você precisaria desenvolver, em vez disso, três instruções `def` em algum outro lugar em seu arquivo, e fora do dicionário no qual as funções devem ser usadas:

```
def f1(): ...
def f2(): ...
def f3(): ...
...
```

```
key = ...
{'already': f1, 'got': f2, 'one': f3}[key]()
```

Esse código também funciona e evita as expressões `lambda`, mas suas instruções `def` podem estar bem distantes entre si em seu arquivo, mesmo que sejam apenas pequenos trechos de código. A *proximidade de código* proporcionada pelas expressões `lambda` é particularmente útil para funções que serão usadas em apenas um contexto – se as três funções aqui não forem úteis em nenhum outro lugar, faz sentido incorporar suas definições dentro do dicionário como expressões `lambda`.

As expressões `lambda` também são úteis em listas de argumentos de função, como uma maneira de colocar em linha definições de função temporárias não utilizadas em nenhuma outra parte de seu programa. Vamos ver exemplos desses outros usos posteriormente neste capítulo, quando estudarmos a função `map`.

Como (não) ofuscar seu código Python

O fato de o miolo de uma expressão `lambda` ter de ser uma única expressão (e não instruções) parece impor sérios limites sobre o volume de lógica que você pode empacotar nela. Contudo, se você souber o que está fazendo, poderá escrever praticamente todas as instruções do Python como uma equivalente baseada em expressão.

Por exemplo, se você quiser imprimir a partir do miolo de uma função `lambda`, basta escrever `sys.stdout.write(str(x)+'\n')`, em vez de `print x`. (Consulte o Capítulo 8, se você tiver se esquecido do motivo.) Analogamente, é possível simular uma instrução `if` combinando operadores booleanos em expressões. A expressão:

```
((a and b) or c)
```

é aproximadamente equivalente a:

```
if a:
    b
else:
    c
```

e é quase a equivalente do Python do operador ternário `a?b:c` da linguagem C. (Para entender o motivo, você precisa ter lido a discussão sobre operadores booleanos no Capítulo 9). Em resumo, os operadores `and` e `or` do Python são de curto-circuito (eles não avaliam o lado direito, se o lado esquerdo determinar o resultado) e sempre retornam o valor da esquerda ou o valor da direita. Em código:

```
>>> t, f = 1, 0
>>> x, y = 88, 99
>>> a = (t and x) or y          # Se for verdadeiro, x
>>> a
88
>>> a = (f and x) or y          # Se for falso, y
>>> a
99
```

Isso funciona, mas contanto que você possa ter certeza de que `x` não será falso também (caso contrário, você receberá sempre `y`). Para simular realmente uma instrução `if` em uma expres-

são, você deve envolver os dois resultados possíveis, de modo a torná-los não-falsos, e então indexar para extrair o resultado no fim:*

```
>>> ((t and [x]) or [y])[0]      # Se for verdadeiro, x
88
>>> ((f and [x]) or [y])[0]      # Se for falso, y
99
>>> (t and f) or y               # Falha: f é falso, pulado
99
>>> ((t and [f]) or [y])[0]      # Funciona: f é retornado de qualquer maneira
0
```

Quando você tiver se atrapalhado digitando isso algumas vezes, provavelmente desejará agrupar para reutilizar:

```
>>> def ifelse(a, b, c): return ((a and [b]) or [c])[0]
...
>>> ifelse(1, 'spam', 'ni')
'spam'
>>> ifelse(0, 'spam', 'ni')
'ni'
```

É claro que aqui você pode obter os mesmos resultados usando, em vez disso, uma instrução `if`:

```
def ifelse(a, b, c):
   if a: return b
   else: return c
```

Mas expressões como essas podem ser colocadas dentro de `lambda` para implementar lógica de seleção:

```
>>> lower = (lambda x, y: (((x < y) and[x]) or [y])[0])
>>> lower('bb', 'aa')
'aa'
>>> lower('aa', 'bb')
'aa'
```

Finalmente, se você precisa realizar loops dentro de uma expressão `lambda`, também pode incorporar coisas como chamadas de `map` e expressões de *abrangência de lista* – ferramentas que conheceremos posteriormente nesta seção:

```
>>> import sys
>>> showall = (lambda x: map(sys.stdout.write, x))
>>> t = showall(['spam\n', 'toast\n', 'eggs\n'])
spam
toast
eggs
```

Mas, agora que mostramos esses truques a você, precisamos pedir para que os utilize apenas como último recurso. Sem o devido cuidado, eles podem levar a um código Python ilegível (também conhecido como *ofuscado*). Em geral, simples é melhor do que complexo, explícito é melhor do que implícito e instruções completas são melhores do que expressões misteriosas. Por outro lado, você pode achá-los úteis, quando usados com moderação.

* Quando escrevemos isto, havia um forte debate em *comp.lang.python* sobre a adição de uma expressão condicional ternária mais direta no Python. Consulte as notas das futuras versões para ver os novos desenvolvimentos nesse sentido. Note que você pode obter quase o mesmo efeito de `and/or` com uma expressão `((falsevalue,truevalue)[condição])`, exceto que isso não dê curto-circuito (os dois resultados possíveis são avaliados a cada vez) e a condição deve ser 0 ou 1.

Expressões lambda aninhadas e escopos

As expressões `lambda` são as principais beneficiárias da pequisa de escopo de função aninhada (a letra E na regra LEGB). No código a seguir, por exemplo, a expressão `lambda` aparece dentro de uma instrução `def` – o caso típico – e, assim, pode acessar o valor que o nome `x` tinha no escopo da função envolvente, quando essa função foi chamada:

```
>>> def action(x):
... return (lambda y: x + y)       # Faz e retorna função.

>>> act = action(99)
>>> act
<function <lambda> at 0x00A16A88>
>>> act(2)
101
```

O que não ilustramos na discussão anterior é que uma expressão `lambda` também tem acesso aos nomes presentes em qualquer expressão `lambda` envolvente. Esse caso é um tanto obscuro, mas imagine se desenvolvêssemos novamente a instrução `def` anterior com uma expressão `lambda`:

```
>>> action = (lambda x (lambda y: x+ y)
>>> act = action(99)
>>> act(3)
102
>>> ((lambda x: (lambda y: x + y))(99))(4)
103
```

Aqui, a estrutura `lambda` aninhada produz uma função que faz uma função quando chamada. Nos dois casos, o código da expressão `lambda` aninhada tem acesso à variável `x` na expressão `lambda` envolvente. Isso funciona, mas é um código bastante complicado. No interesse da legibilidade, geralmente é melhor evitar as expressões `lambda` aninhadas.

APLICANDO FUNÇÕES A ARGUMENTOS

Alguns programas precisam chamar funções arbitrariamente de maneira genérica, sem saber seus nomes nem argumentos antecipadamente. Veremos exemplos de onde isso pode ser útil posteriormente, mas como introdução, a função interna `apply` e a sintaxe de chamada especial fazem o trabalho.

A função interna apply

Você pode chamar funções geradas passando-as como argumentos para `apply`, junto com uma tupla de argumentos:

```
>>> def func(x, y, z): return x + y + z
...
>>> apply(func, (2, 3, 4))
9
>>> f = lambda x, y, z: x + y + z
>>> apply(f, (2, 3, 4))
9
```

A função `apply` simplesmente chama a função passada no primeiro argumento, correspondendo à tupla de argumentos passada com os argumentos esperados da função. Como a lista

Por que isto é relevante: callbacks

Outra aplicação muito comum de `lambda` é na definição de funções de callback em linha para a API de GUI Tkinter. Por exemplo, o código a seguir cria um botão que imprime uma mensagem no console, quando pressionado:

```
import sys
x = Button(
        text ='Press me',
        command=(lambda:sys.stdout.write('Spam\n')))
```

Aqui, a rotina de tratamento de callback é registrada, passando uma função gerada com uma expressão `lambda` para o argumento de palavra-chave `command`. A vantagem de `lambda` sobre `def` é que o código que trata do pressionamento do botão está aqui mesmo – incorporado na chamada de criação do botão.

Na verdade, a expressão `lambda` adia a execução da rotina de tratamento até que o evento ocorra: a chamada de `write` ocorre nos pressionamentos do botão e não quando ele é criado. Como as regras de escopo de função aninhada também se aplicam às expressões `lambda`, elas também são mais fáceis de usar como rotinas de tratamento de callback (a partir do Python 2.2) – elas vêem automaticamente os nomes na função em que são escritas e não exigem mais padrões passados. Isso é particularmente útil para acessar o argumento de instância especial `self`, que é uma variável local nas funções de método de classe envolventes:

```
class MyGui:
    def makewidgets(self):
        Button(command=(lambda: self.display("spam")))
    def display(self, message):
        ...mensagem de uso...
```

Nas versões anteriores, até `self` tinha que ser passado com padrões. Mais informações sobre classes aparecem na Parte VI, e mais informações sobre Tkinter, na Parte VIII.

de argumentos é passada como uma tupla (isto é, uma estrutura de dados), ela pode ser construída em tempo de execução por um programa.*

O poder real de `apply` é que ela não precisa saber com quantos argumentos uma função está sendo chamada. Por exemplo, você pode usar lógica de `if` para fazer uma seleção em um conjunto de funções e listas de argumentos, e usar `apply` para chamar qualquer uma delas:

```
if <test>:
    action, args = func1, (1,)
else:
    action, args = func2, (1, 2, 3)
. . .
apply(action, args)
```

Em geral, a função `apply` é útil sempre que você não pode prever a lista de argumentos antecipadamente. Se seu usuário seleciona uma função arbitrária por intermédio de uma interface com o usuário, por exemplo, talvez seja impossível incorporar o código de uma chamada de

* Cuidado para não confundir `apply` com `map`, o assunto da próxima seção. `apply` executa uma única chamada de função, passando argumentos para o objeto função apenas uma vez. Em vez disso, `map` chama uma função várias vezes, para cada item de uma seqüência.

função ao escrever seu script. Basta construir a lista de argumentos com operações de tupla e chamar indiretamente por meio de `apply`:

```
>>> args = (2, 3) + (4,)
>>> args
(2, 3, 4)
>>> apply (func, args)
9
```

Passando argumentos de palavra-chave

A chamada de `apply` também aceita um terceiro argumento opcional, onde você pode passar um dicionário representando argumentos de palavra-chave a serem passados para a função:

```
>>> def echo(*args, **kwargs): print args, kwargs

>>> echo(1, 2, a=3, b=4)
(1, 2) {'a': 3, 'b': 4}
```

Isso nos permite construir argumentos posicionais e de palavra-chave em tempo de execução:

```
>>> pargs = (1, 2)
>>> kargs = {'a':3, 'b':4}
>>> apply(echo, pargs, kargs)
(1, 2) {'a': 3, 'b': 4}
```

Sintaxe de chamada do tipo apply

O Python também permite obter o mesmo efeito de uma chamada de `apply` com uma sintaxe especial, a qual espelha a sintaxe de argumentos arbitrários em cabeçalhos de `def`, que conhecemos no Capítulo 13. Por exemplo, supondo que os nomes desse exemplo ainda sejam conforme atribuído anteriormente:

```
>>> apply(func, args)                # Tradicional: tupla
9
>>> func(*args)                      # Nova sintaxe tipo apply
9
>>> echo(*pargs, **kargs)            # Dicionários de palavra-chave também
(1, 2) {'a': 3, 'b': 4}
>>> echo(0, *pargs, **kargs)         # Normal, *tupla, **dicionário
(0, 1, 2) {'a': 3, 'b': 4}
```

Essa sintaxe de chamada especial é mais recente do que a função `apply`. Não há nenhuma vantagem óbvia da sintaxe em relação a uma chamada de `apply` explícita, fora sua simetria com cabeçalhos de `def` e alguns toques de tecla a menos.

FUNÇÕES DE MAPEAMENTO SOBRE SEQÜÊNCIAS

Uma das coisas mais comuns que os programas fazem com listas e outras seqüências é aplicar uma operação a cada item e coletar os resultados. Por exemplo, a atualização de todos os contadores em uma lista pode ser feita facilmente com um loop `for`:

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
```

```
...     updated.append(x + 10)          # Soma 10 a cada item.
...
>>> updated
[11, 12, 13, 14]
```

Como essa é uma operação muito comum, o Python fornece uma função interna que faz a maior parte do trabalho para você. A função `map` aplica uma função passada a cada item de um objeto seqüência, e retorna uma lista contendo todos os resultados de chamada de função. Por exemplo:

```
>>> def inc(x): return x + 10           # função a ser executada
...
>>> map(inc, counters)                  # Coleta os resultados.
[11, 12, 13, 14]
```

Apresentamos a função `map` como uma ferramenta para percorrer loop paralelo, no Capítulo 10, onde passamos `None` para o argumento da função para formar pares com os itens. Aqui, fazemos um uso melhor dela, passando uma função real a ser aplicada em cada item da lista – `map` chama `inc` em cada item da lista e coleta todos os valores de retorno em uma lista.

Como a função `map` espera que uma função seja passada, ela também é um dos lugares onde as expressões `lambda` normalmente aparecem:

```
>>> map((lambda x: x + 3), counters)    # Expressão de função
[4, 5, 6, 7]
```

Aqui, a função soma 3 a cada item da lista `counters`. Como essa função não é necessária em nenhum outro lugar, ela foi escrita em linha como uma expressão `lambda`. Como esses usos de `map` são equivalentes aos loops `for`, com um pequeno código extra, você mesmo poderia desenvolver um utilitário de mapeamento geral:

```
>>> def mymap(func, seq):
...     res = []
...     for x in seq: res.append(func(x))
...     return res
...
>>> map(inc, [1, 2, 3])
[11, 12, 13]
>>> mymap(inc, [1, 2, 3])
[11, 12, 13]
```

Entretanto, como `map` é uma função interna, ela está sempre disponível, sempre funciona da mesma forma e tem algumas vantagens de desempenho (em resumo, é mais rápida do que uma instrução `for`). Além disso, a função `map` pode ser usada de maneiras mais avançadas do que o que foi mostrado. Por exemplo, dados vários argumentos de seqüência, ela envia os itens extraídos da seqüência em paralelo, como argumentos distintos para a função:

```
>>> pow(3, 4)
81
>>> map(pow, [1, 2, 3], [2, 3, 4])      # 1**2, 2**3, 3**4
[1, 8, 81]
```

Aqui, a função `pow` recebe dois argumentos em cada chamada – um de cada seqüência passada para `map`. Embora também pudéssemos simular essa generalidade, não há nenhuma razão óbvia para isso, pois a função `map` é interna e rápida.

FERRAMENTAS DE PROGRAMAÇÃO FUNCIONAL

A função `map` é a representante mais simples de uma classe de funções internas do Python usadas para *programação funcional* – o que, de modo geral, significa apenas ferramentas que aplicam funções a seqüências. Suas parentes filtram itens com base em uma função de teste (`filter`) e aplicam funções a pares de itens e resultados correntes (`reduce`). Por exemplo, a chamada de `filter` a seguir extrai itens maiores do que zero de uma seqüência:

```
>>> range(-5, 5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> filter((lambda x: x > 0), range(-5, 5))
[1, 2, 3, 4]
```

Os itens da seqüência para os quais a função retorna verdadeiro são adicionados na lista de resultados. Assim como a função `map`, ela é aproximadamente equivalente a um loop `for`, mas é interna e rápida:

```
>>> res = []
>>> for x in range(-5, 5):
...     if x > 0:
...         res.append(x)
...
>>> res
[1, 2, 3, 4]
```

Aqui estão duas chamadas de `reduce` calculando a soma e o produto de itens em uma lista:

```
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

Em cada etapa, `reduce` passa a soma ou produto corrente, junto com o próximo item da lista, para a função `lambda` passada. Por padrão, o primeiro item da seqüência inicializa o valor de partida. Aqui está o loop `for` equivalente à primeira delas, com a adição incorporada no código dentro do loop:

```
>>> L = [1,2,3,4]
>>> res = L[0]
>>> for x in L[1:]:
...     res = res + x
...
>>> res
10
```

Se isso despertou seu interesse, veja também o módulo interno `operator`, o qual fornece funções que correspondem às expressões internas e, assim, é útil para alguns usos de ferramentas funcionais:

```
>>> import operator
>>> reduce(operator.add, [2, 4, 6])          # + baseado em função
12
>>> reduce((lambda x, y: x + y), [2, 4, 6])
12
```

Alguns observadores também poderiam ampliar o conjunto de ferramentas de programação funcional no Python, incluindo `lambda`, `apply` e abrangências de lista (discutidas na próxima seção).

ABRANGÊNCIA DE LISTA

Como as operações de mapeamento sobre seqüências e a coleta de resultados são tarefas muito comuns no desenvolvimento em Python, a versão 2.0 da linguagem apresentou um novo recurso – a expressão de *abrangência de lista* – que pode tornar isso ainda mais simples do que usar `map` e `filter`. Tecnicamente, esse recurso não está vinculado às funções, mas o deixamos para este ponto do livro porque normalmente é melhor entendido pela analogia com as alternativas baseadas em função.

Fundamentos da abrangência de lista

Vamos ver um exemplo que demonstra os fundamentos. A função interna `ord`, do Python, retorna o código ASCII inteiro de um único caractere:

```
>>> ord('s')
115
```

A função interna `chr` é o oposto – ela retorna o caractere de um inteiro em código ASCII. Agora, suponha que queiramos coletar os códigos ASCII de *todos* os caracteres em uma string completa. Talvez a estratégia mais elementar seja usar um loop `for` simples e anexar os resultados em uma lista:

```
>>> res = []
>>> for x in 'spam':
... res.append(ord(x))
...
>>> res
[115, 112, 97, 109]
```

Agora que conhecemos a função `map`, podemos obter resultados semelhantes com uma única chamada de função, sem ter de gerenciar construção de lista no código:

```
>>> res = map(ord, 'spam')              # Aplica função na seqüência.
>>> res
[115, 112, 97, 109]
```

Mas, do Python 2.0 em diante, obtemos os mesmos resultados a partir de uma expressão de abrangência de lista:

```
>>> res = [ord(x) for x in 'spam']      # Aplica expressão na seqüência.
>>> res
[115, 112, 97, 109]
```

As abrangências de lista coletam os resultados da aplicação de uma expressão arbitrária em uma seqüência de valores e os retornam em uma nova lista. Sintaticamente, as abrangências de lista são colocadas entre colchetes (para lembrá-lo que elas constróem uma lista). Em sua forma simples, dentro dos colchetes você escreve uma expressão que nomeia uma variável, seguida do que parece um cabeçalho de loop `for` que nomeia a mesma variável. O Python coleta os resultados da expressão para cada iteração de loop implícito.

O efeito do exemplo, até aqui, é semelhante ao loop `for` manual e à chamada de `map`. Contudo, as abrangências de lista tornam-se mais úteis quando queremos aplicar uma expressão arbitrária a uma seqüência:

```
>>> [x ** 2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Aqui, coletamos os quadrados dos números 0 a 9. Para fazer um trabalho semelhante com uma chamada de `map`, provavelmente inventaríamos uma pequena função para implementar a operação de elevar ao quadrado. Como não precisaríamos dessa função em outros lugares, ela normalmente seria escrita em linha, com uma expressão `lambda`:

```
>>> map((lambda x: x**2), range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Esse código executa a mesma tarefa e exige apenas alguma digitação a mais do que a abrangência de lista equivalente. Contudo, para tipos de expressões mais avançados, as abrangências de lista freqüentemente serão menores para você digitar. A próxima seção mostra o motivo.

Adicionando testes e loops aninhados

As abrangências de lista são mais gerais do que o mostrado até aqui. Por exemplo, você pode escrever uma cláusula `if` após a instrução `for`, para adicionar lógica de seleção. Abrangências de lista com cláusulas `if` podem ser consideradas análogas à função interna `filter` da seção anterior – elas pulam itens de seqüência para os quais a cláusula `if` não é verdadeira. Aqui estão dois esquemas que selecionam números ímpares de 0 a 4. Assim como a função `map`, `filter` inventa uma pequena função `lambda` para a expressão de teste. Para comparação, o loop `for` equivalente também é mostrado aqui:

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]

>>> filter((lambda x: x % 2 == 0), range(5))
[0, 2, 4]

>>> res = []
>>> for x in range(5):
...     if x % 2 == 0: res.append(x)
...
>>> res
[0, 2, 4]
```

Todos eles estão usando módulo (resto de divisão) para detectar números ímpares: se não houver resto após dividir um número por dois, ele deve ser ímpar. Aqui, a chamada de `filter` também não é muito maior do que a abrangência de lista. Entretanto, a *combinação* de uma cláusula `if` com uma expressão arbitrária proporciona às abrangências de lista o efeito de uma função `filter` e de uma função `map`, em uma única expressão:

```
>>> [x**2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

Desta vez, coletamos os quadrados dos números ímpares de 0 a 9 – o loop `for` pula os números para os quais a cláusula `if` anexada à direita é falsa e a expressão à esquerda calcula os quadrados. A chamada de `map` equivalente exigiria mais trabalho de nossa parte: teríamos que combinar seleções de `filter` com a iteração de `map`, produzindo uma expressão perceptivelmente mais complexa:

```
>>> map((lambda x: x**2), filter((lambda x: x % 2 == 0), range(10)))
[0, 4, 16, 36, 64]
```

Na verdade, as abrangências de lista são ainda mais gerais. Você pode escrever loops `for` aninhados e cada um pode ter um teste `if` associado. A estrutura geral das abrangências de lista é a seguinte:

```
[ expressão for destino1 in seqüência1 [if condição]
            for destino2 in seqüência2 [if condição] ...
            for destinoN in seqüênciaN [if condição] ]
```

Quando cláusulas `for` são aninhadas dentro de uma abrangência de lista, elas funcionam como instruções de loop `for` aninhadas equivalentes. Por exemplo, o código a seguir:

```
>>> res = [x+y for x in [0,1,2] for y in [100,200,300]]
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

tem o mesmo efeito das instruções equivalentes significativamente mais prolixas:

```
>>> res = []
>>> for x in [0,1,2]:
...     for y in [100,200,300]:
...         res.append(x+y)
...
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Embora as abrangências de lista construam uma lista, lembre-se de que elas podem iterar sobre qualquer tipo de seqüência. Aqui está um trecho de código semelhante que percorre strings, em vez de listas de números, e assim coleta resultados da concatenação:

```
>>> [x+y for x in 'spam' for y in 'SPAM']
['sS', 'sP', 'sA', 'sM', 'pS', 'pP', 'pA', 'pM',
'aS', 'aP', 'aA', 'aM', 'mS', 'mP', 'mA', 'mM']
```

Finalmente, aqui está uma abrangência de lista muito mais complexa. Ela ilustra o efeito de seleções `if` anexadas em cláusulas `for` aninhadas:

```
>>> [(x,y) for x in range(5) if x%2 == 0 for y in range(5) if y%2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Essa expressão permuta números ímpares de 0 a 4, com números pares de 0 a 4. A cláusula `if` filtra os itens em cada iteração da seqüência. Aqui está o código baseado em instruções equivalente – ele aninha as cláusulas `for` e `if` da abrangência de lista uma dentro da outra para derivar as instruções equivalentes. O resultado é mais longo, mas talvez mais claro:

```
>>> res = []
>>> for x in range(5):
...     if x % 2 == 0:
...         for y in range(5):
...             if y % 2 == 1:
...                 res.append((x, y))
...
>>> res
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

O código equivalente com `map` e `filter` seria extremamente complexo e aninhado; portanto, nem mesmo tentaremos mostrá-lo aqui. Deixaremos seu desenvolvimento como exercício para os mestres Zen, ex-programadores de LISP e para os dementes.

Compreendendo as abrangências de lista

Com tal generalidade, as abrangências de lista podem tornar-se rapidamente bem incompreensíveis, especialmente quando aninhadas. Por isso, nosso conselho normalmente seria usar loops `for` simples ao se começar a usar o Python e a função `map` na maioria dos outros casos (a menos que elas fiquem complexas demais). A regra "Mantenha simples" se aplica aqui, como sempre. Um código conciso é um objetivo muito menos importante do que um código legível.

Entretanto, atualmente existe uma vantagem de desempenho significativa na complexidade extra, nesse caso: com base em testes executados no Python 2.2, as chamadas de `map` são aproximadamente duas vezes mais rápidas do que os loops `for` equivalentes e, normalmente, as abrangências de lista são muito ligeiramente mais rápidas do que a função `map`. Essa diferença na velocidade se deve ao fato de que a função `map` e as abrangências de lista são executadas na velocidade da linguagem C dentro do interpretador, em vez de percorrer código de loop `for` do Python dentro da PVM.

Por que isto é relevante: abrangências de lista e a função map

Aqui está um exemplo mais realista das abrangências de lista e da função `map` em ação. Lembre-se de que o método de arquivo `readlines` retorna linhas com seu caractere de fim de linha `\n` no final:

```
>>> open('myfile').readlines()
['aaa\n', 'bbb\n', 'ccc\n']
```

Se você não quiser o caractere de fim de linha, pode fracionar todas as linhas em um único passo, com uma abrangência de lista ou com uma chamada de `map`:

```
>>> [line[:-1] for line in open('myfile').readlines()]
['aaa', 'bbb', 'ccc']
>>> [line[:-1] for line in open('myfile')]
['aaa', 'bbb', 'ccc']
>>> map((lambda line: line[:-1]), open('myfile'))
['aaa', 'bbb', 'ccc']
```

Os dois últimos códigos usam *iteradores de arquivo* (basicamente, isso significa que você não precisa de uma chamada de método para pegar todas as linhas em contextos de iteração como esses). A chamada de `map` é apenas ligeiramente mais longa do que as abrangências de lista, mas nenhuma delas precisa gerenciar construção de lista de resultados explicitamente.

As abrangências de lista também podem ser usadas como uma espécie de operação de projeção de coluna. A API de banco de dados SQL padrão do Python retorna resultados de consulta como uma lista de tuplas – a lista é a tabela, as tuplas são linhas e os itens nas tuplas são valores de coluna, muito parecido com a lista a seguir:

```
listoftuple = [('bob', 35, 'mgr'), ('mel', 40, 'dev')]
```

Um loop `for` poderia extrair todos os valores de uma coluna selecionada manualmente, mas a função `map` e as abrangências de lista podem fazer isso em uma única etapa e mais rápido:

```
>>> [age for (name, age, job): age), listoftuple)
[35, 40]
>>> map((lambda (name, age, job): age), listoftuple)
[35, 40]
```

Os dois códigos utilizam atribuição de tupla para desempacotar tuplas de linha na lista. Consulte outros livros e recursos para obter mais informações sobre a API de banco de dados do Python.

Como os loops `for` tornam a lógica mais explícita, os recomendamos em geral por causa da simplicidade. É importante conhecer a função `map` e especialmente as abrangências de lista, se a velocidade de seu aplicativo for uma consideração importante. Além disso, como a função `map` e as abrangências de lista são expressões, elas podem aparecer sintaticamente em lugares que as instruções de loop `for` não podem, como nos miolos de funções `lambda`, dentro de literais de lista e dicionário e muito mais. Contudo, você deve manter suas chamadas de `map` e abrangências de lista simples. Para tarefas mais complexas, use instruções completas, em vez disso.

FUNÇÕES GERADORAS E ITERADORES

É possível escrever funções que podem ser retomadas após enviarem um valor de volta. Tais funções são conhecidas como *geradoras*, pois geram uma seqüência de valores com o passar do tempo. Ao contrário das funções normais que retornam um valor e saem, as funções geradoras suspendem e retomam sua execução automaticamente, e indicam o ponto de geração de valor. Por isso, freqüentemente elas são uma alternativa útil para calcular uma série inteira de valores antecipadamente e para salvar e restaurar estado manualmente em classes.

A principal diferença de código entre as funções geradoras e normais é que as geradoras produzem um valor, em vez de retornar um – a instrução `yield` suspende a função e envia um valor de volta para quem fez a chamada, mas mantém estado suficiente para permitir que a função retome a partir de onde parou. Isso permite que as funções produzam uma série de valores com o passar do tempo, em vez de de calculá-los todos de uma vez, e os enviem de volta em algo como uma lista.

As funções geradoras estão ligadas à noção de protocolos iteradores no Python. Em resumo, as funções que contêm uma instrução `yield` são compiladas de forma especial como geradoras. Quando chamadas, elas retornam um objeto gerador que suporta a interface de objeto iterador.

Os objetos iteradores, por sua vez, definem um método `next`, o qual retorna o próximo item na iteração ou lança uma exceção especial (`StopIteration`) no final da iteração. Os iteradores são buscados com a função interna `iter`. Os loops `for` do Python usam esse protocolo de interface de iteração para percorrer uma seqüência (ou geradora de seqüência), caso o protocolo seja suportado. Se não for, em vez disso o loop `for` retrocede repetidamente na indexação das seqüências.

Exemplo de função geradora

As funções geradoras e os iteradores são recursos avançados da linguagem; portanto, consulte os manuais de biblioteca do Python para ver a história completa das funções geradoras.

Para ilustrar os fundamentos, contudo, o código a seguir define uma função geradora que pode ser usada para gerar os quadrados de uma série de números com o passar do tempo:*

```
>>> def gensquares(N):
...     for i in range(N):
...         yield i ** 2           # Retoma aqui posteriormente.
```

Essa função produz um valor e, então, retorna para quem a chamou, cada vez que percorre o loop. Quando ela é retomada, seu estado anterior é restaurado e o controle assume novamente,

* As funções geradoras estão disponíveis nas versões do Python posteriores a 2.2. Na versão 2.2, elas devem ser chamadas com uma instrução de importação especial da forma `from __future__ import generators`. (Consulte o Capítulo 18 para ver mais informações sobre essa forma de instrução.) Os iteradores já estavam disponíveis na versão 2.2, em grande parte porque o protocolo subjacente não exigia a nova palavra-chave sem compatibilidade com versões anteriores `yield`.

imediatamente após a instrução `yield`. Por exemplo, quando usada como seqüência em um loop `for`, o controle retomará a função após sua instrução `yield`, cada vez que percorrer o loop:

```
>>> for i in gensquare(5):        # Retoma a função.
...     print i, ':',             # Imprime o último valor produzido.
...
0 : 1 : 4 : 9 : 16 :
>>>
```

Para finalizar a geração de valores, as funções usam uma instrução `return` sem nenhum valor, ou simplesmente abandonam no fim do miolo da função. Se você quiser ver o que está acontecendo dentro do loop `for`, chame a função geradora diretamente:

```
>>> x = gensquares(10)
>>> x
<generator object at 0x0086C378>
```

Você recebe um objeto gerador que suporta o *protocolo iterador* – ele tem um método `next` que inicia a função ou a retoma a partir de onde produziu um valor pela última vez:

```
>>> x.next()
0
>>> x.next()
1
>>> x.next()
4
```

Os loops `for` trabalham com funções geradoras da mesma forma – chamando o método `next` repetidamente, até que uma exceção seja capturada. Se o objeto no qual vai ser feita a iteração não suporta esse protocolo, os loops `for` usam o protocolo de indexação para iterar.

Note que, nesse exemplo, também poderíamos simplesmente construir de uma vez toda a lista de valores produzidos:

```
>>> def buildsquares(n):
...     res = []
...     for i in range(n): res.append(i**2)
...     return res
...
>>> for x in buildsquares(5): print x, ':',
...
0 : 1 : 4 : 9 : 16 :
```

Quanto a isso, poderíamos simplesmente usar qualquer uma das técnicas de loop `for`, função `map` ou abrangência de lista:

```
>>> for x in [n**2 for n in range(5)]:
...     print x, ':',
...
0 : 1 : 4 : 9 : 16 :
>>> for x in map((lambda x:x**2), range(5)):
...     print x, ':',
...
0 : 1 : 4 : 9 : 16 :
```

Entretanto, especialmente quando as listas de resultados são grandes ou quando é necessária muita computação para produzir cada valor, as geradoras permitem que as funções evitem fazer todo o trabalho antecipadamente. Elas distribuem o tempo exigido para produzir a série de valores entre as iterações do loop. Além disso, para usos mais avançados, as geradoras for-

necem uma alternativa mais simples para salvar manualmente o estado entre as iterações em objetos classe (mais informações sobre classes aparecerão posteriormente na Parte VI). Com as geradoras, as variáveis de função são salvas e restauradas automaticamente.

Iteradores e tipos internos

Os tipos de dados internos são projetados para produzir objetos iteradores em resposta à função interna `iter`. Os iteradores de dicionário, por exemplo, produzem itens de lista de chave em cada iteração:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> x = iter(D)
>>> x.next()
'a'
>>> x.next()
'c'
```

Além disso, por sua vez, todos os contextos de iteração, incluindo os loops `for`, as chamadas de `map` e as compreensões de lista, são projetados para chamar automaticamente a função `iter`, para ver se o protocolo é suportado. É por isso que você pode fazer um loop pelas chaves de um dicionário sem chamar seu método `keys`, percorrer as linhas de um arquivo sem chamar `readlines` ou `xreadlines` etc.:

```
>>> for key in D:
...     print key, D[key]
...
a 1
c 3
b 2
```

Para iteradores de arquivo, o Python 2.2 usa simplesmente o resultado do método de arquivo `xreadlines`. Esse método retorna um objeto que carrega linhas do arquivo de acordo com a demanda e lê por grupos de linhas, em vez de carregar o arquivo inteiro todo de uma vez:

```
>>> for line in open('temp.txt'):
...     print line,
...
Tis but
a flesh wound.
```

Também é possível implementar objetos arbitrários com classes, as quais são compatíveis com o protocolo iterador e, portanto, podem ser usadas em loops `for` e em outros contextos de iteração. Tais classes definem um método `__iter__` especial que retorna um objeto iterador (preferido em relação ao método de indexação `__getitem__`). Entretanto, isso está bem além dos objetivos deste capítulo. Consulte a Parte VI para obter mais informações sobre classes em geral e o Capítulo 21 para ver um exemplo de classe que implementa o protocolo iterador.

CONCEITOS DE PROJETO DE FUNÇÃO

Quando você começa a usar funções, se depara com escolhas a respeito de como reunir componentes – por exemplo, como decompor uma tarefa em funções (*coesão*), como as funções devem se comunicar (*acoplamento*) etc. Parte disso cai na categoria da análise e projeto estruturados. Aqui estão algumas dicas gerais para iniciantes em Python:

Acoplamento: use argumentos para entradas e retorno para saídas. Geralmente, você deve se esforçar para tornar uma função independente do que está fora dela. Os argumentos e as instruções de retorno freqüentemente são as melhores maneiras de isolar dependências externas.

Acoplamento: use variáveis globais apenas quando forem realmente necessárias. As variáveis globais (isto é, nomes no módulo envolvente) normalmente são uma maneira insatisfatória para as funções se comunicarem. Elas podem criar dependências e problemas de sincronismo que tornam os programas difíceis de depurar e alterar.

Acoplamento: não altere argumentos mutáveis a não ser que quem fez a chamada espere isso. As funções também podem alterar partes de objetos mutáveis passados. Mas, assim como nas variáveis globais, isso implica em muito acoplamento entre quem fez a chamada e quem foi chamado, o que pode tornar uma função específica demais e frágil.

Coesão: cada função deve ter um só propósito unificado. Quando bem projetada, cada uma de suas funções deve fazer apenas uma coisa – algo que você possa resumir em uma frase declarativa simples. Se essa frase é muito abrangente (por exemplo, "esta função implementa meu programa inteiro") ou contém muitas conjunções (por exemplo, "esta função dá aumento para os funcionários *e* envia um pedido de pizza"), talvez você queira pensar em dividi-la em funções separadas e mais simples. Caso contrário, não haverá como reutilizar o código existente por trás das etapas misturadas em tal função.

Tamanho: cada função deve ser relativamente pequena. Isto resulta naturalmente do objetivo da coesão, mas se suas funções começarem a abranger várias páginas em sua tela, provavelmente é hora de dividir. Particularmente porque, em primeiro lugar, o código Python é conciso; uma função longa ou profundamente aninhada freqüentemente é um sintoma de problemas de projeto. Mantenha-a simples e curta.

A Figura 14-1 resume as maneiras como as funções podem comunicar-se com o mundo externo. As entradas podem vir dos itens no lado esquerdo e os resultados podem ser enviados em qualquer uma das formas à direita. Normalmente, alguns projetistas de função usam apenas argumentos para entradas e instruções de retorno para saídas.

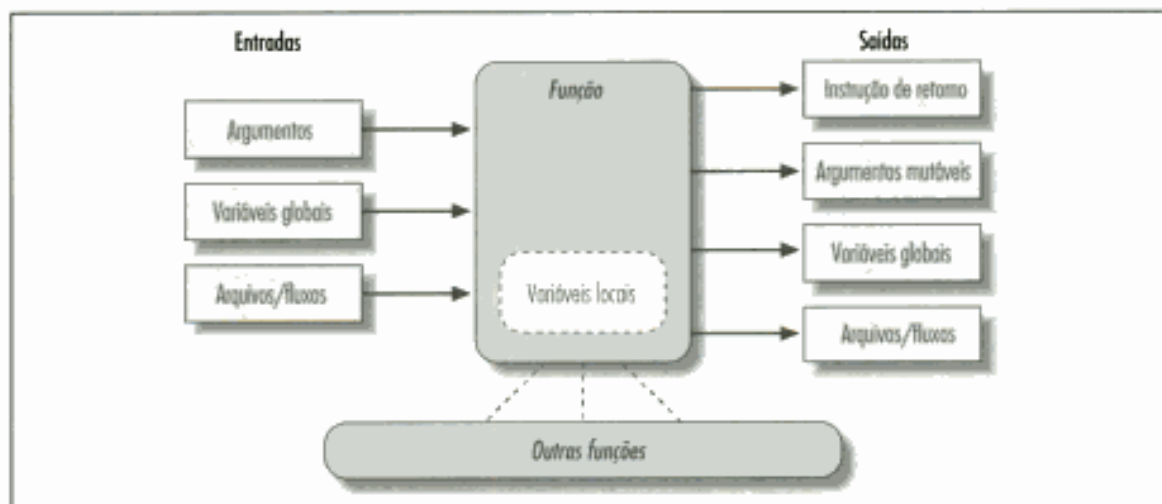



Figura 14-1 Ambiente de execução de função.

Existem muitas exceções, incluindo o suporte para programação orientada a objetos do Python – conforme você verá na Parte VI, as classes do Python *dependem* da alteração de um objeto mutável passado. As funções de classe configuram atributos de um argumento passado

automaticamente, chamado `self`, para alterar informações de estado de acordo com o objeto (por exemplo, `self.name='bob'`). Além disso, se não são usadas classes, freqüentemente as variáveis globais são a melhor maneira para funções em módulos manterem o estado entre chamadas. Se forem previstos, tais efeitos colaterais não são perigosos.

As funções são objetos: chamadas indiretas

Como as funções do Python são objetos em tempo de execução, você pode escrever programas que as processam genericamente. Os objetos função podem ser atribuídos, passados para outras funções, armazenados em estruturas de dados etc., como se fossem números ou strings simples. Vimos alguns desses usos em exemplos anteriores. Os objetos função exportam uma operação especial – eles podem ser chamados listando-se argumentos entre parênteses após uma expressão de função. Mas as funções pertencem à mesma categoria geral que outros objetos.

Por exemplo, não há nada realmente especial quanto ao nome usado em uma instrução `def`: trata-se apenas de uma variável atribuída no escopo corrente, como se tivesse aparecido à esquerda de um sinal `=`. Depois que uma instrução `def` é executada, o nome da função é uma referência para um objeto. Você pode atribuir esse objeto novamente para outros nomes e chamá-lo por meio de qualquer referência – e não apenas pelo nome original:

```
>>> def echo(message):               # Eco atribuído a um objeto função.
...     print message
...
>>> x = echo                         # Agora x faz referência a ele também.
>>> x('Hello world!')                # Chama o objeto adicionando ().
Hello world!
```

Como os argumentos são passados pela atribuição de objetos, é fácil passar funções para outras funções, como argumentos. Quem é chamado pode, então, chamar a função passada apenas adicionando argumentos entre parênteses:

```
>>> def indirect(func, arg):
...     func(arg)                    # Chama os objetos adicionando ().
...
>>> indirect(echo, 'Hello jello!')   # Passa função para uma função.
Hello jello!
```

Você pode até colocar objetos função em estruturas de dados, como se eles fossem inteiros ou strings. Como os tipos compostos do Python podem conter qualquer tipo de objeto, também não há nenhum caso especial aqui:

```
>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>> for (func, arg) in schedule:
...      func(arg)
...
Spam!
Ham!
```

Esse código apenas percorre a lista schedule, chamando a função `echo` com um argumento a cada vez. A ausência de declarações de tipo do Python contribui para que ele seja uma linguagem de programação incrivelmente flexível. Observe a atribuição de desempacotamento de tupla no cabeçalho do loop `for`, apresentada no Capítulo 8.

PROBLEMAS DAS FUNÇÕES

Aqui estão alguns dos problemas mais capciosos que você pode não esperar. Todos são obscuros e alguns começaram a desaparecer da linguagem completamente nas versões recentes, mas a maioria têm atrapalhado os usuários iniciantes.

Os nomes locais são detectados estaticamente

O Python classifica os nomes atribuídos em uma função como locais, por padrão. Eles ficam no escopo da função e só existem enquanto ela está em execução. O que não dissemos antes é que o Python detecta os locais estaticamente, ao compilar o código da instrução `def`, em vez de notar as atribuições quando elas acontecem em tempo de execução. Isso leva a uma das esquisitices mais comuns postadas por iniciantes no newsgroup do Python.

Normalmente, um nome que não é atribuído em uma função é pesquisado no módulo envolvente:

```
>>> X = 99
>>> def selector():                    # X usado mas não atribuído
...     print X                        # X encontrado no escopo global
...
>>> selector()
99
```

Aqui, o `X` na função se transforma no `X` do módulo externo. Mas observe o que acontece se você adiciona uma atribuição a `X` após a referência:

```
>>> def selector():
...     print X          # Ainda não existe!
...     X = 88           # X classificado como nome local (por toda parte)
...                      # Também pode acontecer se for "import X", "def X",...
>>> selector()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in selector
UnboundLocalError: local variable 'X' referenced before assignment
```

Você recebe um erro de nome indefinido, mas o motivo é sutil. O Python lê e compila esse código quando ele é digitado interativamente ou importado de um módulo. Ao compilar, o Python vê a atribuição para `X` e decide que `X` será um nome local em todas as partes da função. Mas, posteriormente, quando a função é realmente executada, a atribuição ainda não aconteceu, quando a instrução `print` é executada; portanto, o Python diz que você está usando um nome indefinido. De acordo com suas regras de nome, ele está certo. O `X` local é usado antes de ser atribuído. Na verdade, qualquer atribuição no miolo de uma função torna um nome local. Importações, `=`, instruções def aninhadas, classes aninhadas etc, todas são suscetíveis a esse comportamento.

O problema ocorre porque os nomes atribuídos são tratados como locais por toda parte em uma função, e não apenas após as instruções onde eles são atribuídos. Na realidade, o exemplo anterior é ambíguo, na melhor das hipóteses: você queria imprimir o `X global` e depois criar um `X local` ou esse é um erro de programação genuíno? Como o Python trata `X` como local por toda parte, é um erro. Mas se você quiser realmente imprimir `X`, precisa declará-lo em uma instrução global:

```
>>> def selector():
...     global X               # Força X a ser global (por toda parte).
...     print X
```

```
...     X = 88
...
>>> selector()
99
```

Lembre-se, contudo, de que isso significa que a atribuição também altera o `X global` e não um `X local`. Dentro de uma função, você não pode usar versões locais e globais do mesmo nome simples. Se você quiser realmente imprimir o global e depois configurar um local de mesmo nome, importe o módulo envolvente e qualifique para obter a versão global:

```
>>> X = 99
>>> def selector():
...     import __main__         # Importa o módulo envolvente.
...     print __main__.X        # Qualifica para obter a versão global do nome.
...     X = 88                  # X desqualificado classificado como local.
...     print X                 # Imprime a versão local do nome.
...
>>> selector()
99
88
```

A qualificação (`a parte .X`) busca um valor de um objeto espaço de nome. O espaço de nome interativo é um módulo chamado `__main__`; portanto, `__main__.X` atinge a versão global de `X`. Se isso não estiver claro, consulte a Parte V.*

Padrões e objetos mutáveis

Os valores de argumento padrão são avaliados e salvos quando a instrução `def` é executada e não quando a função resultante é chamada. Internamente, o Python salva um objeto por argumento padrão, vinculado à própria função.

Normalmente é isso que você quer. Como os padrões são avaliados no momento da execução de `def`, isso permite que você salve valores do escopo envolvente, se necessário. Mas como os padrões mantêm um objeto entre chamadas, você precisa tomar cuidado com a alteração de padrões mutáveis. Por exemplo, a função a seguir usa uma lista vazia como valor padrão e depois a altera no local, sempre que a função é chamada:

```
>>> def saver(x=[]):            # Salva um objeto lista
...     x.append(1)             # Altera o mesmo objeto a cada vez!
...     print x
...
>>> saver([2])                  # Padrão não usado
[2, 1]
>>> saver()                     # Padrão usado
[1]
>>> saver()                     # Cresce a cada chamada!
[1, 1]
>>> saver()
[1, 1, 1]
```

Alguns vêem esse comportamento como um recurso – como os argumentos padrão mutáveis mantêm seu estado entre chamadas de função, eles podem ter algumas das mesmas funções

* O Python aprimorou bastante essa história, emitindo a mensagem de erro "unbound local" (local desvinculado) para esse caso, mostrado no exemplo de listagem (ela simplesmente gerava um erro de nome genérico). Em geral, contudo, esse problema ainda está presente.

das variáveis de função locais *estáticas* da linguagem C. De certo modo, eles funcionam como variáveis globais, mas seus nomes são locais para a função e, assim, não entram em conflito com nomes de outras partes de um programa.

Contudo, para a maioria dos observadores, isso parece um problema, especialmente na primeira vez que o encontram. Existem maneiras melhores de manter o estado entre chamadas no Python (por exemplo, usando classes, que serão discutidas na Parte VI).

Além disso, os padrões mutáveis são difíceis de lembrar (e de entender). Eles dependem do sincronismo da construção do objeto padrão. No exemplo, há apenas um objeto lista a cada vez que a função é chamada; portanto, a lista cresce a cada nova anexação; ela não é reconfigurada como vazia em cada chamada.

Se esse não é o comportamento desejado, basta fazer cópias do padrão no início do miolo da função ou mover a expressão de valor padrão para o miolo da função. Contanto que o valor resida no código que é executado cada vez que a função é executada, você obterá um novo objeto a cada vez:

```
>>> def saver(x=None):
...     if x is None:               # Nenhum argumento passado?
...         x = []                  # Executa código para fazer uma nova lista.
...     x.append(1)                 # Altera o novo objeto lista
...     print x
...
>> saver([2])
[2, 1]
>>> saver()                         # Não cresce aqui
[1]
>>> saver()
[1]
```

A propósito, a instrução `if` do exemplo *quase* poderia ser substituída pela atribuição `x = x or []`, que tira proveito do fato de que o operador `or` do Python retorna um de seus objetos operando: se nenhum argumento fosse passado, `x` teria `None` como padrão; portanto, o operador `or` retornaria a nova lista vazia à direita.

Entretanto, isso não é exatamente igual. Quando uma lista vazia é passada, a expressão `or` faria a função estender e retornar uma lista recentemente criada, em vez de estender e retornar a lista passada, como na versão anterior. (A expressão torna-se `[] or []`, que é avaliada como a nova lista vazia à direita. Consulte a seção "Testes de verdade", no Capítulo 9, se você não se lembra do motivo.) Os requisitos de programa reais podem exigir um ou outro comportamento.

Funções sem retornos

Nas funções do Python, as instruções `return` (e `yield`) são opcionais. Quando uma função não retorna um valor explicitamente, ela termina quando o controle chega ao fim de seu miolo. Tecnicamente, todas as funções retornam um valor. Se você não fornecer um retorno, sua função retornará o objeto `None` automaticamente:

```
>>> def proc(x):
...     print x                     # Nenhum retorno é um retorno None.
...
>>> x = proc('testing 123...')
testing 123...
>>> print x
None
```

Funções como essa, sem uma instrução `return`, são o equivalente do Python do que se chamam "procedures" em algumas linguagens. Normalmente, elas são executadas como uma instrução e o resultado `None` é ignorado, pois fazem seu trabalho sem calcular um resultado útil.

É interessante saber disso, pois o Python não o informará se você tentar usar o resultado de uma função que não retorna outra função. Por exemplo, atribuir o resultado de um método de lista `append` não produzirá um erro, mas você receberá de volta `None` e não a lista modificada:

```
>>> list = [1, 2, 3]
>>> list = list.append(4)        # append é uma "procedure".
>>> print list                   # append altera a lista no local.
None
```

Conforme mencionado na seção "Problemas de desenvolvimento comuns" do Capítulo 11, tais funções fazem seu trabalho como um efeito colateral e, normalmente, são projetadas para serem executadas como uma instrução e não como uma expressão.

EXERCÍCIOS DA PARTE IV

Vamos começar a desenvolver programas mais sofisticados nestes exercícios. Verifique as soluções no Apêndice B e comece a escrever seu código em arquivos de módulo. Você não vai querer digitar novamente esses exercícios desde o início, se cometer um erro.

1. *Os fundamentos*. No prompt interativo do Python, escreva uma função que imprima seu único argumento na tela e chame-a interativamente, passando uma variedade de tipos de objeto: string, inteiro, lista, dicionário. Em seguida, tente chamá-la sem passar nenhum argumento. O que acontece? O que acontece quando você passa dois argumentos?
2. *Argumentos*. Escreva uma função chamada `adder` em um arquivo de módulo Python. A função `adder` deve aceitar dois argumentos e retornar a soma (ou concatenação) dos dois. Em seguida, adicione código no final do arquivo para chamar a função com uma variedade de tipos de objeto (duas strings, duas listas, dois números em ponto flutuante) e execute esse arquivo como um script a partir da linha de comando do sistema. Você precisa imprimir os resultados da instrução de chamada para vê-los em sua tela?
3. *varargs*. Generalize a função `adder` que você escreveu no último exercício para calcular a soma de um número arbitrário de argumentos e altere as chamadas para passar mais ou menos de dois. De que tipo é a soma do valor de retorno? (Dicas: um fracionamento como `S[:0]` retorna uma seqüência vazia do mesmo tipo de `S` e a função interna `type` pode testar tipos. Mas consulte os exemplos de `min` no Capítulo 13 para ver uma estratégia mais simples.) O que acontece se você passa argumentos de tipos diferentes? E quanto à passagem de dicionários?
4. *Palavras-chave*. Altere a função `adder` do Exercício 2 para aceitar e somar três argumentos: `def adder(good, bad, ugly)`. Agora, forneça valores padrão para cada argumento e experimente chamar a função interativamente. Tente passar um, dois, três e quatro argumentos. Em seguida, tente passar argumentos de palavra-chave. A chamada `adder(ugly=1, good=2)` funciona? Por quê? Finalmente, generalize a nova função `adder` para aceitar e somar um número arbitrário de argumentos de palavra-chave, de forma muito parecida com o Exercício 3, mas você precisará iterar sobre um dicionário e não sobre uma tupla. (Dica: o método `dict.keys()` retorna uma lista que você pode percorrer com um loop `for` ou `while`.)

5. Escreva uma função chamada `copyDict(dict)` que copie seu argumento de dicionário. Ela deve retornar um novo dicionário com todos os itens em seu argumento. Use o método de dicionário `keys` para iterar (ou, no Python 2.2, percorra as chaves de um dicionário sem chamar `keys`). Copiar seqüências é fácil (`X[:]` faz uma cópia de nível superior). Isso também funciona para dicionários?

6. Escreva uma função chamada `addDict(dict1, dict2)` que calcule a união de dois dicionários. Ela deve retornar um novo dicionário, com todos os seus itens em seus dois argumentos (supostos como sendo dicionários). Se a mesma chave aparecer nos dois argumentos, fique à vontade para escolher o valor de um dos dois. Teste sua função escrevendo-a em um arquivo e executando o arquivo como um script. O que acontece se você passa listas, em vez de dicionários? Como você poderia generalizar sua função para tratar desse caso também? (Dica: veja a função interna `type` usada anteriormente.) A ordem dos argumentos passados importa?

7. *Mais exemplos de correspondência de argumentos*. Primeiro, defina as seis funções a seguir (interativamente ou em um arquivo de módulo que possa ser importado):

```
def f1(a, b): print a, b                # Args normais

def f2(a, *b): print a, b               # varargs posicionais

def f3(a, **b): print a, b              # varargs de palavra-chave

def f4(a, *b, **c): print a, b, c       # Modos misturados

def f5(a, b=2, c=3): print a, b, c      # Padrões

def f6(a, b=2, *c): print a, b, c       # Padrões e varargs posicionais
```

Agora, teste as seguintes chamadas interativamente e tente explicar cada resultado. Em alguns casos, você provavelmente precisará recorrer ao algoritmo de correspondência mostrado no Capítulo 13. Você acha que misturar modos de correspondência é uma boa idéia em geral? Você consegue imaginar casos em que isso seria útil?

```
>>> f1(1, 2)
>>> f1(b=2, a=1)

>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, x=2, y=3)

>>> f5(1)
>>> f5(1, 4)

>>> f6(1)
>>> f6(1, 3, 4)
```

8. *Números primos revisitados*. Lembre-se do trecho de código que vimos no Capítulo 10, o qual determina de forma simplificada se um número positivo é primo:

```
x = y / 2                           # Para algum y > 1
while x > 1:
    if y % x == 0:                  # Resto
         print y, 'has factor', x
         break                      # Pula a cláusula else
    x = x-1
else:                               # Saída normal
    print y, 'is prime'
```

Empacote esse código como uma função reutilizável em um arquivo de módulo e adicione algumas chamadas para sua função no final de seu arquivo. Enquanto está nisso, substitua o operador / da primeira linha por //, para fazê-lo tratar também de números de ponto flutuante e ficar imune à alteração da divisão "real" planejada para o operador / no Python 3.0, conforme descrito no Capítulo 4. O que você pode fazer quanto aos números negativos e 0 e 1? E quanto ao aumento da velocidade do código? Suas saídas devem ser como as seguintes:

```
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
```

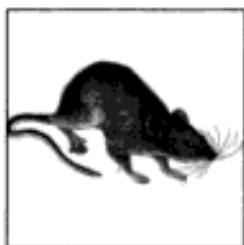
9. *Abrangência de lista.* Escreva um código para construir uma nova lista contendo as raízes quadradas de todos os números desta lista: `[2, 4, 9, 16, 25]`. Escreva isso primeiro como um loop `for`, em seguida, como uma chamada de `map` e, finalmente, como uma abrangência de lista. Use a função `sqrt` do módulo interno `math` para fazer o cálculo (isto é, importe `math` e escreva `math.sqrt(x)`). Das três, de qual estratégia você gosta mais?

V

Módulos

A Parte V explora os módulos do Python. Os módulos são pacotes de nomes que normalmente correspondem aos arquivos-fonte e servem como bibliotecas de ferramentas para uso em outros arquivos e programas. Apresentamos os módulos muito cedo (na Parte I), como uma maneira de manter e executar código. Aqui, completaremos os detalhes restantes sobre esse assunto e estudaremos alguns tópicos mais avançados relacionados aos módulos, como as importações de pacote (diretório).

15

Módulos: o Panorama Geral

Este capítulo inicia nosso exame dos *módulos* do Python, a unidade de organização de programa de nível mais alto, a qual empacota código de programa e dados para reutilização. Em termos concretos, os módulos normalmente correspondem aos arquivos de programa do Python (ou extensões da linguagem C). Cada arquivo é um módulo e os módulos importam outros módulos para usar os nomes neles definidos. Os módulos são processados com duas novas instruções e uma função interna importante:

`import`
: Permite que um cliente (importador) busque um módulo como um todo

`from`
: Permite que os clientes busquem nomes específicos de um módulo

`reload`
: Fornece uma maneira de recarregar o código de um módulo sem interromper o Python.

Apresentamos os fundamentos do módulo no Capítulo 3 e, desde então, os estivemos usando. A Parte V começa expandindo os conceitos básicos de módulo e, depois, passa a explorar a utilização mais avançada dos módulos. Este primeiro capítulo começa com uma visão geral da função dos módulos na estrutura global do programa. No próximo capítulo e nos seguintes, entraremos nos detalhes do desenvolvimento existentes por trás da teoria.

No processo, mostraremos os detalhes dos módulos que omitimos até aqui: recarregamentos, os atributos `__name__` e `__all__`, importações de pacote etc. Como os módulos e as classes são, na verdade, apenas espaços de nome especiais, também formalizaremos os conceitos de espaço de nome aqui.

POR QUE USAR MÓDULOS?

Os módulos proporcionam uma maneira fácil de organizar componentes em um sistema, servindo como pacotes de nomes. A partir de uma perspectiva abstrata, os módulos têm pelo menos três funções:

Reutilização de código

Conforme vimos no Capítulo 3, os módulos nos permitem salvar código permanentemente em arquivos. Ao contrário do código digitado no prompt interativo do Python, o qual desaparece quando você sai dele, o código que fica em arquivos de módulo é persistente – ele pode ser recarregado e novamente executado quantas vezes forem necessárias. Mais objetivamente, os módulos são locais para definir nomes (ou atributos) que podem ser referenciados por clientes externos.

Particionamento do espaço de nome do sistema

Os módulos também são a unidade de organização de programa de nível mais alto no Python. Basicamente, eles são apenas pacotes de nomes. Os módulos encerram nomes em pacotes auto-suficientes que evitam conflitos de nome – você nunca pode ver um nome em outro arquivo, a menos que o importe explicitamente. Na verdade, tudo "vive" em um módulo: o código que você executa e os objetos que cria são sempre englobados implicitamente por um módulo. Por isso, os módulos são uma ferramenta natural para agrupar componentes de sistema.

Implementar serviços ou dados compartilhados

A partir de uma perspectiva funcional, os módulos também são úteis para implementar componentes compartilhados em um sistema e, por isso, só exigem uma única cópia. Por exemplo, se você precisa fornecer um objeto global usado por mais de uma função ou arquivo, pode escrevê-lo em um módulo que será importado por muitos clientes.

Contudo, para entendermos realmente a função dos módulos em um sistema Python, precisamos nos desviar por uns instantes e explorar a estrutura geral de um programa em Python.

ARQUITETURA DE PROGRAMA EM PYTHON

Até aqui neste livro, amenizamos parte da complexidade em nossas descrições dos programas em Python. Na prática, os programas normalmente têm mais do que apenas um arquivo: para todos os scripts, menos os mais simples, seus programas assumirão a forma de sistemas de vários arquivos. Além disso, ainda que você mesmo possa escrever apenas um arquivo, quase certamente acabará usando arquivos externos que alguém já escreveu.

Esta seção apresenta a arquitetura geral dos programas em Python – a maneira como você divide um programa em uma coleção de arquivos-fonte (também conhecidos como módulos) e vincula as partes em um todo. No processo, também definiremos os principais conceitos dos módulos do Python, importações e atributos de objeto.

Como estruturar um programa

Geralmente, um programa em Python consiste em vários arquivos de texto contendo instruções da linguagem. O programa é estruturado como um arquivo *de nível superior* principal, junto com zero ou mais arquivos complementares, conhecidos no Python como *módulos*.

No Python, o arquivo de nível superior contém o fluxo de controle principal de seu programa – o arquivo que você executa para executar seu aplicativo. Os arquivos de módulo são bibliotecas de ferramentas, usadas para reunir componentes utilizados pelo arquivo de nível superior e, possivelmente, em outras partes. Os arquivos de nível superior utilizam ferramentas definidas em arquivos de módulo, e os módulos utilizam ferramentas definidas em outros módulos. No Python, um arquivo *importa* um módulo para ter acesso às ferramentas definidas por ele. Além disso, as ferramentas definidas por um módulo são conhecidas como *atributos*

– nomes de variável vinculados a objetos, como funções. Em última análise, importamos módulos e acessamos seus atributos para usar suas ferramentas.

Importações e atributos

Vamos tornar isso um pouco mais concreto. A Figura 15-1 esboça a estrutura de um programa em Python composto de três arquivos: *a.py*, *b.py* e *c.py*. O arquivo *a.py* é escolhido como sendo o de nível superior. Ele será um arquivo de texto simples, com instruções, que é executado de cima para baixo, quando ativado. Os arquivos *b.py* e *c.py* são módulos. Eles também são arquivos de texto simples, com instruções, mas normalmente não são executados diretamente. Em vez disso, os módulos normalmente são importados por outros arquivos que desejam utilizar as ferramentas neles definidas.

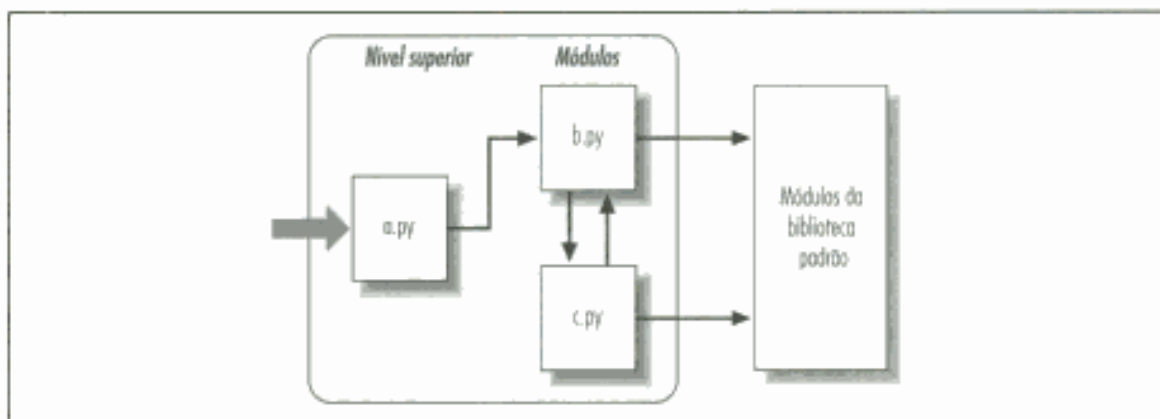



Figura 15-1 Arquitetura de programa.

Por exemplo, suponha que o arquivo *b.py* da Figura 15-1 defina uma *função* chamada `spam`, para uso externo. Conforme aprendemos na Parte IV, *b.py* conteria uma instrução `def` do Python para gerar a função, a qual é executada posteriormente pela passagem de zero ou mais valores entre parênteses, após o nome da função:

```
def spam(text):
    print text, 'spam'
```

Agora, se *a.py* quisesse usar `spam`, esse arquivo poderia conter instruções do Python, como as seguintes:

```
import b
b.spam('gumby')
```

A primeira das duas, uma instrução `import` do Python, dá ao arquivo *a.py* acesso a tudo que está definido no arquivo *b.py*. Isso significa, aproximadamente: "carregue o arquivo *b.py* (a não ser que ele já esteja carregado) e me dê acesso a todos os seus atributos por meio do nome `b`". As instruções `import` (e, conforme você verá posteriormente, `from`) executam e carregam outro arquivo em tempo de execução. No Python, o vínculo de módulo entre os arquivos não é solucionado até que essas instruções `import` sejam executadas.

A segunda dessas instruções chama a função `spam` definida no módulo `b`, usando notação de atributo de objeto. O código `b.spam` significa: "busque o valor do nome `spam` que vive dentro do objeto `b`". Acontece que essa é uma função que pode ser chamada, em nosso exemplo; portanto, passamos uma string entre parênteses (`'gumby'`). Se você digitar realmente esses arquivos e executar *a.py*, as palavras "gumby spam" serão impressas.

Em geral, você verá a notação `objeto.atributo` em todos os scripts Python – a maioria dos objetos tem atributos úteis que são buscados com o operador ".". Algumas coisas podem ser chamadas, como as funções, e outras são simples valores de dados que fornecem propriedades para o objeto (por exemplo, o nome de uma pessoa).

A noção de importação também é geral por todo o Python. Qualquer arquivo pode importar ferramentas de qualquer outro arquivo. Por exemplo, o arquivo *a.py* pode importar *b.py* para chamar sua função, mas *b.py* também poderia importar *c.py* para utilizar ferramentas diferentes lá definidas. Os encadeamentos de importação podem ter a profundidade que você quiser: neste exemplo, o módulo `a` pode importar `b`, que pode importar `c`, que pode importar `b` novamente e assim por diante.

Além de servir como uma estrutura de organização de mais alto nível, os módulos (e os pacotes de módulo, descritos no Capítulo 17) também são o nível mais alto de *reutilização de código* no Python. Desenvolvendo-se componentes em arquivos de módulo, eles tornam-se úteis no programa original, assim como em qualquer outro programa que você possa escrever. Por exemplo, se após desenvolvermos o programa da Figura 15-1, descobrirmos que a função `b.spam` é uma ferramenta de propósito geral, podemos reutilizá-la em um programa completamente diferente; basta importar o arquivo *b.py* novamente, a partir dos arquivos do outro programa.

Módulos da biblioteca padrão

Observe a parte da direita da Figura 15-1. Alguns dos módulos que seu programa importará são fornecidos pelo próprio Python e não por arquivos que você escreverá. O Python já vem com uma grande coleção de módulos utilitários, conhecidos como *biblioteca padrão*.

Essa coleção, de aproximadamente 200 módulos na última contagem, contém suporte independente de plataforma para tarefas de programação comuns: interfaces de sistema operacional, persistência de objeto, correspondência de padrão de texto, scripts de rede e Internet, construção de GUI e muito mais. Nenhum deles faz parte da linguagem Python em si, mas podem ser usados importando-se os módulos apropriados em qualquer instalação de Python padrão.

Neste livro, você vai ver alguns dos módulos da biblioteca padrão em ação nos exemplos, mas para um panorama completo, consulte o manual de referência da biblioteca, disponível em sua instalação do Python (ela está no IDLE e na entrada do botão Iniciar do Python, no Windows) ou on-line, no endereço *http://www.python.org*.

Como existem muitos módulos, essa é a única maneira de ter uma idéia das ferramentas que estão disponíveis. Você também pode encontrar materiais da biblioteca Python em livros disponíveis comercialmente, mas os manuais são gratuitos, podem ser vistos em qualquer navegador da Web (eles vêm em formato HTML) e são atualizados sempre que o Python é relançado.

COMO AS IMPORTAÇÕES FUNCIONAM

A seção anterior discutiu a importação de módulos, sem realmente explicar o que acontece quando você faz isso. Como as importações são o centros da estrutura de um programa no Python, esta seção entra em mais detalhes sobre essa operação para tornar o processo menos abstrato.

Alguns programadores de C gostam de comparar a operação de importação de módulo do Python com uma instrução `#include` daquela linguagem, mas na verdade não deveriam – no Python, as importações não são apenas inserções textuais de um arquivo em outro. Elas são realmente operações em tempo de execução que realizam três etapas distintas na primeira vez que um arquivo é importado por um programa:

1. *Localizar* o arquivo do módulo.
2. *Compilá-lo* no código de byte (se necessário)
3. *Executar* o código do módulo para construir os objetos que ele define.

Todas essas três etapas só são realizadas na primeira vez que um módulo é importado durante a execução de um programa. As importações posteriores do mesmo módulo ignoram todas elas e simplesmente buscam o objeto módulo já carregado na memória. Para entendermos melhor as importações de módulo, vamos explorar cada uma dessas etapas por sua vez.

1. Localizar

Primeiramente, o Python precisa localizar o arquivo de módulo referenciado pela sua instrução de importação. Observe que a instrução de importação, no exemplo da seção anterior, nomeia o arquivo sem um sufixo *.py* e sem seu caminho de diretório. Ela diz apenas `import b`, em vez de algo como `import c:\dir1\b.py`. As instruções de importação omitem os detalhes do caminho e do sufixo, como esses, de propósito; você só pode listar um nome simples.* Em vez disso, o Python usa um *caminho de pesquisa de módulo* padrão para localizar o arquivo de módulo correspondente a uma instrução de importação.

O caminho de pesquisa de módulo

Em muitos casos, você pode contar com a natureza automática do caminho de pesquisa de importação de módulo, e não precisa configurá-lo. Contudo, se você quiser importar arquivos além dos limites de diretório definidos pelo usuário, precisará saber como o caminho de pesquisa funciona para personalizá-lo. A grosso modo, o caminho de pesquisa de módulo do Python é composto automaticamente como a concatenação dos seguintes componentes principais:

1. O diretório de base do arquivo de nível superior.
2. Diretórios de `PYTHONPATH` (se estiver configurada).
3. Diretórios da biblioteca padrão.
4. O conteúdo de todos os arquivos *.pth* (se estiverem presentes).

O primeiro e o terceiro desses itens são definidos automaticamente. Como o Python pesquisa a concatenação deles do primeiro para o último, o segundo e o quarto podem ser usados para estender o caminho de pesquisa de módulo para incluir os seus diretórios. Aqui está como o Python usa cada um desses componentes do caminho:

Diretório de base

O Python procura o arquivo importado primeiro no diretório de base. Dependendo de como você está executando o código, esse é o diretório que contém o arquivo de nível superior de seu programa ou o diretório no qual está trabalhando interativamente. Como ele é sempre pesquisado primeiro, se um programa estiver localizado inteiramente em

* Na verdade, é sintaticamente inválido incluir os detalhes do caminho e do sufixo em uma importação. No Capítulo 17, conheceremos as *importações de pacote*, as quais permitem que as instruções de importação incluam parte do caminho de diretório no início de um arquivo, como um conjunto de nomes separados por pontos-finais. Entretanto, as importações de pacote ainda contam com o caminho de pesquisa de módulo normal para localizar o diretório mais à esquerda em um caminho de pacote. Elas também não podem usar qualquer sintaxe de diretório específica de uma plataforma na instrução de importação; tal sintaxe só funciona no caminho de pesquisa. Note também que os problemas de caminho de pesquisa de arquivo de módulo não são tão relevantes quando você utiliza *executáveis congelados* (discutidos no Capítulo 2); normalmente, eles incorporam código de byte na imagem binária.

um único diretório, todas as suas importações funcionarão automaticamente, sem exigir nenhuma configuração de caminho.

Diretórios de `PYTHONPATH`

Em seguida, o Python pesquisa todos os diretórios listados na configuração de sua variável de ambiente `PYTHONPATH`, da esquerda para a direita (supondo que você a tenha configurado). Em resumo, `PYTHONPATH` é simplesmente configurada como uma lista de nomes definidos pelo usuário e específicos da plataforma de diretórios que contêm arquivos de código Python. Adicione todos os diretórios dos quais você queira importar. O Python usa sua configuração para estender o caminho de pesquisa de módulo.

Como o Python pesquisa primeiro o diretório de base, você só precisa fazer essa configuração para importar arquivos além dos limites de diretório – isto é, para importar um arquivo armazenado em um diretório diferente do arquivo que o importa. Na prática, você provavelmente fará essa configuração quando começar a escrever programas significativos. Contudo, quando estiver começando, se você salvar todos os seus arquivos de módulo no diretório em que estiver trabalhando (isto é, o diretório de base), suas importações funcionarão sem a necessidade dessa configuração.

Diretórios da biblioteca padrão

Depois, o Python pesquisará automaticamente os diretórios onde os módulos da biblioteca padrão estão instalados em sua máquina. Como eles são sempre pesquisados, normalmente não precisam ser adicionados em sua variável de ambiente `PYTHONPATH`.

Diretórios de arquivos .pth

Finalmente, um recurso relativamente novo do Python permite aos usuários adicionarem diretórios válidos no caminho de pesquisa de módulo, simplesmente listando-os, um por linha, em um arquivo de texto cujo nome termine com o sufixo *.pth* (de "path" – caminho). Esses arquivos de configuração de caminho são um recurso um tanto avançado, relacionado à instalação, o qual não discutiremos completamente aqui.

Em resumo, um arquivo de texto de nomes de diretório, colocado em um diretório apropriado, pode ter mais ou menos a mesma função da configuração da variável de ambiente `PYTHONPATH`. Por exemplo, um arquivo chamado *myconfig.pth* pode ser colocado no nível superior do diretório de instalação do Python no Windows (por exemplo, em C:\Python22), para estender o caminho de pesquisa de módulo. O Python adicionará os diretórios listados em cada linha do arquivo, do primeiro ao último, próximos ao final da lista do caminho de pesquisa de módulo. Como são baseados em arquivos, em vez das configurações de shell, os arquivos de caminho também podem se aplicar a todos os usuários de uma instalação, em vez de a apenas um usuário ou shell.

Esse recurso é mais sofisticado do que descreveremos aqui. Recomendamos aos iniciantes usarem `PYTHONPATH` ou um único arquivo *.pth* e, além disso, somente se precisarem fazer importação entre diretórios. Consulte o manual da biblioteca Python para ver mais detalhes sobre esse recurso, especialmente sua documentação do módulo de biblioteca padrão site.

Consulte também o Apêndice A para ver exemplos de maneiras comuns de estender seu caminho de pesquisa de módulo com `PYTHONPATH` ou com arquivos *.pth* em várias plataformas. Dependendo de sua plataforma, mais diretórios também podem ser adicionados automaticamente no caminho de pesquisa de módulo. Na verdade, esta descrição do caminho de pesquisa de módulo é precisa, mas genérica. A configuração exata do caminho de pesquisa pode mudar de acordo com as plataformas e versões do Python.

Por exemplo, o Python pode adicionar uma entrada para o *diretório de trabalho corrente* – o diretório a partir do qual você executou seu programa – no caminho de pesquisa, após os diretórios de `PYTHONPATH` e antes das entradas da biblioteca padrão. Ao se executar a partir de uma linha de comando, o diretório de trabalho corrente pode não ser o *diretório de base* de seu arquivo de nível superior – o diretório onde seu arquivo de programa reside. (Consulte o Capítulo 3 para obter mais informações sobre linhas de comando.) Como o diretório de trabalho corrente pode variar sempre que seu programa é executado, você normalmente não deve depender de seu valor para propósitos de importação.

A lista sys.path

Se você quiser ver como o caminho está realmente configurado em sua máquina, sempre pode inspecionar o caminho de pesquisa de módulo, como ele é conhecido no Python, imprimindo a lista interna `sys.path` (isto é, o atributo `path` do módulo interno `sys`). Essa lista de strings de nome de diretório do Python é o caminho de pesquisa real. Nas importações, o Python pesquisa cada diretório da lista, da esquerda para a direita.

Na realidade, `sys.path` *é* o caminho de pesquisa de módulo. Ela é configurada pelo Python na inicialização do programa, usando os quatro componentes de caminho que acabamos de descrever. O Python intercala automaticamente nessa lista todas as configurações de caminho de `PYTHONPATH` e de arquivo *.pth* que você fez e sempre configura a primeira entrada para identificar o diretório de base do arquivo de nível superior, possivelmente como uma string vazia.

O Python expõe essa lista por dois bons motivos. Primeiramente, a lista proporciona uma maneira de verificar as configurações de caminho de pesquisa que você fez – se você não ver suas configurações em algum lugar nessa lista, precisará conferir seu trabalho. Em segundo lugar, se você souber o que está fazendo, essa lista também proporciona uma maneira para os scripts personalizarem seus caminhos de pesquisa manualmente. Conforme verá posteriormente nesta parte, modificando a lista `sys.path`, você pode modificar o caminho de pesquisa para todas as importações futuras. Entretanto, essas alterações duram apenas enquanto o script existe; `PYTHONPATH` e os arquivos *.pth* são maneiras mais permanentes de modificar o caminho.*

Seleção de arquivo de módulo

Lembre-se de que os *sufixos* de nome de arquivo (por exemplo, *.py*) são omitidos intencionalmente nas instruções de importação. O Python escolhe o primeiro arquivo que encontrar no caminho de pesquisa, que corresponda ao nome importado. Por exemplo, uma instrução de importação da forma `import b` poderia carregar:

- O arquivo-fonte *b.py*
- O arquivo de código de byte *b.pyc*
- Um diretório chamado *b*, para importações de pacote
- Um módulo de extensão C (por exemplo, `b.so` no Linux)
- Uma imagem na memória, para executáveis congelados

* Contudo, alguns programas precisam realmente alterar `sys.path`. Os scripts executados em servidores da Web, por exemplo, normalmente são executados como o usuário "nobody" (ninguém) para limitar o acesso à máquina. Como esses scripts normalmente não podem depender de "nobody" para configurar `PYTHONPATH` de alguma maneira específica, eles freqüentemente configuram `sys.path` manualmente para incluir os diretórios-fonte exigidos, antes de executar quaisquer instruções de importação.

- Uma classe Java, no sistema Jython
- Um componente de arquivo zip, usando o módulo zipimport

Alguns módulos da biblioteca padrão são, na verdade, escritos em C. As extensões C, o Jython e as importações de pacote, todos importam mais do que arquivos simples. Contudo, para os importadores, a diferença no tipo de arquivo carregado é completamente transparente, tanto ao importar quanto ao buscar atributos de módulo. Escrever `import b` obtém o que quer que seja o módulo `b`, de acordo com seu caminho de pesquisa de módulo, e `b.attr` busca um item no módulo, seja ele uma variável do Python ou uma função vinculada da linguagem C. Alguns módulos padrão que usaremos neste livro, por exemplo, são escritos em C e não em Python; seus clientes não precisam se preocupar.

Se você tiver *b.py* e *b.so* em diretórios diferentes, o Python sempre carregará o que estiver no primeiro diretório (mais à esquerda) em seu caminho de pesquisa de módulo, durante a pesquisa da esquerda para a direita de `sys.path`. Mas o que acontece se *b.py* e *b.so* existem no *mesmo* diretório? O Python segue uma ordem de escolha padrão, mas não é garantido que ela permaneça a mesma com o passar do tempo. Em geral, você não deve depender do tipo de arquivo que o Python escolherá dentro de determinado diretório – torne seus nomes de módulo distintos ou use configuração de caminho de pesquisa de módulo para tornar a seleção de módulo mais evidente. Também é possível redefinir grande parte do que uma operação de importação faz no Python, com o que são conhecidos como *ganchos de importação*. Esses ganchos podem ser usados para que as importações façam coisas úteis, como carregar arquivos de repositórios de arquivo zip, descriptografar etc. (na verdade, o Python 2.3 inclui um módulo padrão `zipimport` que permite aos arquivos serem importados diretamente de repositórios de arquivo zip). Normalmente, contudo, as importações funcionam conforme descrito nesta seção. O Python também suporta a noção de arquivos de código de byte otimizados *.pyo*, criados e executados com o flag de linha de comando `-O` da linguagem. Como eles são executados de forma apenas ligeiramente mais rápida do que os arquivos *.pyc* normais (normalmente, 5% mais rápido), não são muito utilizados. O sistema Psyco (veja o Capítulo 2) proporciona aumentos de velocidade mais significativos.

2. Compilar (talvez)

Após encontrar um arquivo de código-fonte correspondente à instrução de importação, de acordo com o caminho de pesquisa de módulo, o Python o compila em código de byte, se necessário. (Discutimos o código de byte no Capítulo 2.)

O Python verifica as indicações de tempo do arquivo e pula a etapa de compilação do código-fonte em código de byte, se encontra um arquivo de código de byte *.pyc* que não seja mais antigo do que o arquivo-fonte *.py* correspondente. Além disso, se o Python encontra apenas um arquivo de código de byte no caminho de pesquisa e nenhum código-fonte, ele simplesmente carrega o código de byte diretamente. Por isso, se possível, a etapa de compilação é ignorada, para acelerar a inicialização do programa. Se você alterar o código-fonte, o Python regenerará automaticamente o código de byte na próxima vez que seu programa for executado. Além disso, você pode distribuir um programa apenas com arquivos de código de byte e evitar o envio do código-fonte.

Note que a compilação acontece quando um arquivo está sendo *importado*. Por isso, normalmente você não verá um arquivo de código de byte *.pyc* para o arquivo de nível superior de seu programa, a não ser que ele também seja importado em outro lugar – apenas os arquivos importados deixam para trás um arquivo *.pyc* em sua máquina. O código de byte de arquivos de nível superior é usado internamente e descartado. O código de byte de arquivos importados é salvo em arquivos para acelerar as futuras importações.

Os arquivos de nível superior são freqüentemente designados para serem executados diretamente e não serem importados. Posteriormente, veremos que é possível projetar um arquivo que serve tanto como o código de nível superior de um programa quanto como um módulo de ferramentas para ser importado. Tais arquivos podem ser executados e importados e, assim, geram um arquivo *.pyc*. Para aprender sobre isso, veja a discussão sobre o atributo especial `__name__` e `"__main__"`, no Capítulo 18.

3. Executar

A última etapa de uma operação de importação executa o código de byte do módulo. Todas as instruções que estão no arquivo são executadas por sua vez, de cima para baixo, e todas as atribuições feitas a nomes durante esta etapa geram atributos do objeto módulo resultante. Esta etapa de execução gera todas as ferramentas definidas pelo código do módulo. Por exemplo, as instruções `def` de um arquivo são executadas no momento da importação para criar funções e designam atributos para essas funções dentro do módulo. As funções são chamadas posteriormente no programa, pelos importadores.

Como essa última etapa da importação realmente executa o código do arquivo, se algum código de nível superior em um arquivo de módulo realizar trabalho real, você verá seus resultados no momento da importação. Por exemplo, as instruções `print` de nível superior em um módulo mostram a saída quando o arquivo é importado. As instruções de função `def` simplesmente definem objetos para uso posterior.

Conforme você pode ver, as operações de importação envolvem bastante trabalho – elas pesquisam arquivos, possivelmente executam um compilador e executam código Python. Por padrão, determinado módulo é importado apenas uma vez por processo. As futuras importações pulam todas as três etapas e reutilizam o módulo que já está carregado na memória.*

Conforme você também pode ver, a operação de importação é o centro da arquitetura de programa no Python. Os programas maiores são divididos em vários arquivos, os quais são vinculados pelas importações em tempo de execução. Por sua vez, as importações usam caminhos de pesquisa de módulo para localizar seus arquivos e os módulos definem atributos para uso externo.

Naturalmente, o objetivo das importações e dos módulos é fornecer uma estrutura para seu programa, a qual divide a lógica em componentes de software independentes. O código de um módulo é isolado do código de outro. Na verdade, nenhum arquivo jamais pode ver os nomes definidos em outro, a não ser que sejam executadas instruções `import` explícitas. Para vermos o que tudo isso significa em termos de código real, vamos passar para o Capítulo 16.

* Tecnicamente, o Python mantém os módulos já carregados no dicionário interno `sys.modules` e verifica isso no início de uma operação de importação para saber se o módulo já está carregado. Se você quiser ver quais módulos estão carregados, importe `sys` e imprima `sys.modules.keys()`. Mais informações sobre essa tabela interna aparecem no Capítulo 18.

16

Fundamentos do Desenvolvimento de Módulos

Agora que já vimos as idéias mais amplas existentes por trás dos módulos, vamos passar para um exemplo simples de módulos em ação. Os módulos do Python são fáceis de *criar*; eles são apenas arquivos de código de programa da linguagem, criados com seu editor de textos. Você não precisa escrever sintaxe especial para informar ao Python que está criando um módulo; praticamente qualquer arquivo de texto servirá. Como o Python trata de todos os detalhes da localização e do carregamento de módulos, estes também são fáceis de *usar*. Os clientes simplesmente importam um módulo ou nomes específicos definidos por um módulo e usam os objetos que ele referencia.

CRIAÇÃO DE MÓDULOS

Para definir um módulo, use seu editor de textos para digitar código Python em um arquivo de texto. Os nomes atribuídos no nível superior do módulo tornam-se atributos (nomes associados ao objeto módulo) e são exportados para os clientes usarem. Por exemplo, se digitarmos a instrução `def` a seguir em um arquivo chamado *module1.py* e o importarmos, criaremos um objeto módulo com um atributo – o nome `printer`, que é uma referência para um objeto função:

```
def printer(x):          # Atributo de módulo
    print x
```

Uma palavra sobre nomes de arquivo de módulo: você pode dar praticamente qualquer nome para seus módulos, mas se pretende importá-los, os nomes de arquivo de módulo devem terminar com o sufixo *.py*. Tecnicamente, o sufixo *.py* é opcional para os arquivos de nível superior que serão executados, mas não importados. Porém, adicionar o sufixo em todos os casos torna o tipo do arquivo mais evidente.

Como os nomes de módulo tornam-se variáveis dentro de um programa em Python sem o sufixo *.py*, eles também devem seguir as regras normais de nome de variável que aprendemos no Capítulo 8. Por exemplo, você pode criar um arquivo de módulo chamado *if.py*, mas não pode importá-lo, pois `if` é uma palavra reservada – quando você tentar executar `import if`, obterá um erro de sintaxe. Na verdade, os nomes de arquivos de módulo e de diretórios, usados em importações de pacote, devem obedecer as regras dos nomes de variável apresentadas

no Capítulo 8. Isso se torna uma consideração mais importante para diretórios de pacote; seus nomes não podem conter sintaxe específica de uma plataforma, como espaços.

Quando os módulos são importados, o Python faz o mapeamento do nome de módulo interno para um nome de arquivo externo, adicionando caminhos de diretório no caminho de pesquisa do módulo na frente e um sufixo *.py* ou outra extensão no final. Por exemplo, em última análise, o nome de módulo `M` é mapeado em algum arquivo externo `<diretório>\M.<extensão>` que contém o código de nosso módulo.

UTILIZAÇÃO DE MÓDULOS

Os clientes podem usar o arquivo de módulo que acabamos de escrever, executando instruções `import` ou `from`. Ambas localizam, compilam e executam o código de um arquivo de módulo, se ele ainda não tiver sido carregado. A principal diferença é que `import` busca o módulo como um todo, de modo que você precisa qualificar para buscar seus nomes; em vez disso, `from` busca (ou copia) nomes específicos do módulo.

Vamos ver o que isso significa em termos de código. Todos os exemplos a seguir acabam chamando a função `printer` definida no arquivo de módulo externo *module1.py*, mas de maneiras diferentes.

A instrução import

No primeiro exemplo, o nome `module1` tem dois propósitos diferentes. Ele identifica um arquivo externo a ser carregado e torna-se uma variável no script, a qual referencia o objeto módulo após o arquivo ser carregado:

```
>>> import module1                      # Obtém o módulo como um todo.
>>> module1.printer('Hello world!')     # Qualifica para obter nomes.
Hello world!
```

Como `import` fornece um nome que se refere ao objeto módulo como um todo, devemos passar pelo nome do módulo para buscar seus atributos (por exemplo, `module1.printer`).

A instrução from

Em contraste, como `from` também copia nomes de um arquivo para outro escopo, devemos, em vez disso, usar os nomes copiados diretamente, sem passar pelo módulo (por exemplo, `printer`):

```
>>> from module1 import printer         # Copia uma variável.
>>> printer('Hello world!')             # Não precisa qualificar o nome.
Hello world!
```

A instrução from *

Finalmente, o próximo exemplo usa uma forma especial de `from`: quando usamos `*`, obtemos cópias de *todos* os nomes atribuídos no nível superior do módulo referenciado. Aqui, novamente, usamos o nome copiado e não passamos pelo nome do módulo:

```
>>> from module1 import *               # Copia todas as variáveis.
>>> printer('Hello world!')
Hello world!
```

Tecnicamente, tanto a instrução `import` como `from` executam a mesma operação de importação; `from` simplesmente acrescenta uma etapa de cópia extra.

É isso. Os módulos são muito simples de usar. Mas para dar a você um entendimento melhor do que realmente acontece quando define e utiliza módulos, vamos ver algumas de suas propriedades com mais detalhes.

As importações acontecem apenas uma vez

Uma das perguntas mais comuns que os iniciantes parecem fazer ao usar módulos é: por que minhas importações não continuam funcionando? A primeira importação funciona bem, mas as importações posteriores, durante uma sessão interativa (ou execução de programa), parecem não ter nenhum efeito. Elas não devem ter mesmo e aqui está o motivo.

Os módulos são carregados e executados na primeira instrução `import` ou `from`. Entretanto, a operação de importação só acontece na primeira importação, de propósito – como essa é uma operação dispendiosa, por padrão, o Python faz isso apenas uma vez por processo. As operações de importação posteriores apenas buscam um objeto módulo já carregado.

Como conseqüência, já que o código de nível superior, em um arquivo de módulo, normalmente é executado apenas uma vez, você pode usá-lo para inicializar variáveis. Considere o arquivo *simple.py*, por exemplo:

```
print 'hello'
spam = 1                          # Inicializa variável.
```

Nesse exemplo, as instruções `print` e `=` são executadas na primeira vez que o módulo é importado e a variável `spam` é inicializada no momento da importação:

```
% python
>>> import simple                 # Primeira importação: carrega e
                                  executa o código do arquivo
hello
>>> simple.spam                   # A atribuição produz um atributo.
1
```

Entretanto, a segunda importação e as posteriores não executam novamente o código do módulo, mas apenas buscam o objeto módulo já criado na tabela de módulos interna do Python – a variável `spam` não é reinicializada:

```
>>> simple.spam = 2               # Altera atributo no módulo.
>>> import simple                 # Apenas busca o módulo já carregado.
>>> simple.spam                   # O código não foi executado novamente:
                                    atributo inalterado.
2
```

É claro que, às vezes, você quer realmente que o código de um módulo seja executado de novo. Veremos como fazer isso com a função interna `reload`, posteriormente neste capítulo.

import e from são atribuições

Assim como `def`, `import` e `from` são instruções executáveis e não declarações em tempo de compilação. Elas podem ser aninhadas em testes `if`, aparecer em instruções `def` de função etc., e não são solucionadas nem executadas até que o Python as alcance, enquanto seu programa executa. Também como a instrução `def`, `import` e `from` são atribuições implícitas:

- `import` atribui um objeto módulo inteiro a um único nome.
- `from` atribui um ou mais nomes a objetos de mesmo nome em outro módulo.

Tudo que já dissemos sobre atribuição se aplica também ao acesso a módulo. Por exemplo, os nomes copiados com uma instrução `from` tornam-se referências para objetos compartilhados. Assim como nos argumentos de função, atribuir novamente um nome buscado não tem nenhum efeito sobre o módulo do qual ele foi copiado, mas alterar um *objeto mutável* buscado pode alterá-lo no módulo do qual foi importado. O arquivo *small.py* ilustra isso:

```
x = 1
y = [1, 2]
```

```
% python
>>> from small import x, y  # copia dois nomes.
>>> x = 42                  # Altera apenas o x local
>>> y[0] = 42               # Altera o objeto mutável compartilhado no local
```

Aqui, alteramos um objeto mutável compartilhado que obtivemos com a atribuição de `from`: o nome `y` no importador e o importado fazem referência ao mesmo objeto lista; portanto, sua alteração em um lugar o altera no outro:

```
>>> import small            # Obtém nome de módulo (from não obtém).
>>> small.x                 # O x de small não é o meu x.
1
>>> small.y                 # Mas compartilhamos um objeto mutável alterado.
[42, 2]
```

Na verdade, para ver uma imagem gráfica do que fazem as atribuições de `from`, volte para a Figura 13-2 (passagem de argumentos de função). Substitua mentalmente "quem fez a chamada" e "função" por "importado" e "importador" para ver o que as atribuições de `from` fazem com as referências. É exatamente o mesmo efeito, exceto que aqui estamos tratando com nomes em módulos e não com funções. A atribuição funciona da mesma forma em todos os lugares no Python.

Alterações de nome entre arquivos

Observe, no exemplo anterior, como a atribuição para `x` na sessão interativa altera o nome `x` apenas nesse escopo e não o `x` do arquivo – não há nenhum vínculo de um nome copiado com `from` para o arquivo de onde ele veio. Para alterar um nome global em outro módulo, você deve usar `import`:

```
% python
>>> from small import x, y      # Copia dois nomes.
>>> x = 42                      # Altera apenas meu x

>>> import small                # Obtém nome de módulo.
>>> small.x = 42                # Altera x no outro módulo
```

Como a alteração de variáveis em outros módulos, como essa, é comumente confusa (e freqüentemente uma escolha de projeto ruim), vamos rever novamente essa técnica, posteriormente neste capítulo. A alteração de `y[0]` na sessão anterior altera um objeto e não um nome.

Equivalência de import e from

A propósito, note que também precisamos executar uma instrução `import` no exemplo anterior, após a instrução `from`, para ter acesso ao nome de módulo `small`. A instrução `from` ape-

nas copia nomes de um módulo para outro e não atribui o nome do módulo em si. Pelo menos conceitualmente, uma instrução `from` como a seguinte:

```
from module import name1, name2       # Copia esses dois nomes (apenas).
```

é equivalente a esta seqüência:

```
import module                         # Busca o objeto módulo.
name1 = module.name1                  # Copia nomes pela atribuição.
name2 = module.name2
del module                            # Desfaz-se do nome do módulo.
```

Assim como todas as atribuições, a instrução `from` cria novas variáveis no importador, as quais inicialmente referem-se aos objetos de mesmo nome no arquivo importado. Contudo, apenas os nomes são copiados e não o módulo em si.

ESPAÇOS DE NOME DE MÓDULO

Provavelmente, os módulos são melhor entendidos simplesmente como pacotes de nomes – lugares para definir os nomes que você deseja tornar visíveis para o restante de um sistema. No Python, os módulos são *espaços de nome* – um lugar onde nomes são criados. Os nomes que ficam em um módulo são chamados de seus *atributos*. Tecnicamente, os módulos normalmente correspondem a arquivos e o Python cria um objeto módulo para conter todos os nomes atribuídos no arquivo, mas em termos simples, os módulos são apenas espaços de nome.

Arquivos geram espaços de nome

Então, como os arquivos se transformam em espaços de nome? A história curta é que todo nome que recebe um valor no nível superior de um arquivo de módulo (isto é, não aninhado no miolo de uma função ou de uma classe) torna-se um atributo desse módulo.

Por exemplo, dada uma instrução de atribuição como `X=1` no nível superior de um arquivo de módulo *M.py*, o nome `X` torna-se um atributo de `M`, ao qual podemos nos referir fora do módulo como `M.X`. O nome `X` também torna-se uma variável global para outro código dentro de *M.py*, mas precisamos explicar a noção de carregamento de módulo e escopos um pouco mais formalmente para entendermos o motivo:

As instruções de módulo são executadas na primeira importação. Na primeira vez que um módulo é importado em qualquer parte de um sistema, o Python cria um objeto módulo vazio e executa as instruções presentes no arquivo de módulo, uma após a outra, do início ao final do arquivo.

As atribuições de nível superior criam atributos de módulo. Durante uma importação, as instruções presentes no nível superior do arquivo que atribuem nomes (por exemplo, `=`, `def`) criam atributos do objeto módulo. Os nomes atribuídos são armazenados no espaço de nome do módulo.

Espaço de nome de módulo: atributo `__dict__` ou `dir(M)`. Os espaços de nome de módulo criados pelas importações são dicionários. Eles podem ser acessados por intermédio do atributo interno `__dict__` associado aos objetos de módulo e pode ser inspecionado com a função `dir`. A função `dir` é aproximadamente equivalente à lista de chaves ordenadas do atributo `__dict__` de um objeto, mas inclui nomes herdados de classes, pode não ser completa e é propensa a mudar de uma versão para outra.

Os módulos são um único escopo (local é global). Conforme vimos no Capítulo 13, os nomes no nível superior de um módulo seguem as mesmas regras de referência/atribui-

ção que os nomes em uma função, mas os escopos local e global são o mesmo (ou, mais precisamente, é a regra LEGB, mas sem as camadas de pesquisa L e E). Mas, nos módulos, o *escopo* torna-se um dicionário de atributos de um *objeto* módulo, após o módulo ter sido carregado. Ao contrário das funções (onde o espaço de nome local existe apenas enquanto a função é executada), o escopo de um arquivo de módulo torna-se o espaço de nome de atributos de um objeto módulo e permanece após a importação.

Aqui está uma demonstração dessas idéias. Suponha que criemos o seguinte arquivo de módulo com um editor de textos e o chamemos de *module2.py*:

```
print 'starting to load...'

import sys
name = 42

def func(): pass

class klass: pass

print 'done loading.'
```

Na primeira vez que esse módulo é importado (ou executado como um programa), o Python executa suas instruções do início ao fim. Algumas instruções criam nomes no espaço de nome do módulo, como um efeito colateral, mas outras podem realizar trabalho real, enquanto a importação está ocorrendo. Por exemplo, as duas instruções `print` desse arquivo são executadas no momento da importação:

```
>>> import module2
starting to load...
done loading.
```

Mas, uma vez carregado o módulo, seu escopo torna-se um espaço de nome de atributos no objeto módulo que recebemos de `import` – acessamos os atributos nesse espaço de nome qualificando-os com o nome do módulo envolvente:

```
>>> module2.sys
<module 'sys'>
>>> module2.name
42
>>> module2.func, module2.klass
(<function func at 765f20>, <class klass at 76df60>)
```

Aqui, `sys`, `name`, `func` e `klass` foram todos atribuídos enquanto as instruções do módulo estavam sendo executadas; portanto, eles são atributos após a importação. Vamos falar sobre classes na Parte VI, mas observe o atributo `sys`: as instruções `import` realmente *atribuem* objetos módulo a nomes, e qualquer tipo de atribuição a um nome, no nível superior de um arquivo, gera um atributo de módulo. Internamente, os espaços de nome de módulo são armazenados como objetos dicionário. Na verdade, podemos acessar o dicionário de espaços de nome por meio do atributo `__dict__` do módulo; ele é apenas um objeto dicionário normal, com os métodos usuais:

```
>>> module2.__dict__.keys()
['__file__', 'name', '__name__', 'sys', '__doc__', '__builtins__', 'klass',
'func']
```

Internamente, os nomes que atribuímos no arquivo de módulo tornam-se chaves de dicionário. Alguns dos nomes no espaço de nome do módulo são itens que o Python adiciona para nós. Por exemplo, `__file__` fornece o nome do arquivo do qual o módulo foi carregado e `__name__` fornece seu nome conforme conhecido para os importadores (sem a extensão *.py* e o caminho de diretório).

Qualificação de nome de atributo

Agora que você está se tornando mais familarizado com os módulos, devemos esclarecer a noção de *qualificação* de nome. No Python, você pode acessar atributos em qualquer objeto que tenha atributos, usando a sintaxe de qualificação `objeto.atributo`.

Na realidade, a qualificação é uma expressão que retorna o valor atribuído a um nome de atributo associado a um objeto. Por exemplo, a expressão `module2.sys`, do exemplo anterior, busca o valor atribuído a `sys` em `module2`. Analogamente, se temos um objeto lista interno `L`, `L.append` retorna o método associado à lista.

Então, o que a qualificação do atributo faz com as regras de escopo que estudamos no Capítulo 13? Na verdade, nada: esse é um conceito independente. Quando você usa qualificação para acessar nomes, fornece ao Python um objeto explícito para fazer a busca. A regra LEGB só se aplica a nomes simples, não qualificados. Aqui estão as regras:

Variáveis simples

`X` significa procurar o nome `X` nos escopos correntes (regra LEGB).

Qualificação

`X.Y` significa localizar `X` nos escopos correntes e, em seguida, procurar o atributo `Y` no objeto `X` (não nos escopos).

Caminhos de qualificação

`X.Y.Z` significa pesquisar o nome `Y` no objeto `X` e, em seguida, pesquisar `Z` no objeto `X.Y`.

Generalidade

A qualificação funciona em todos os objetos com atributos: módulos, classes, tipos da linguagem C etc.

Na Parte VI, veremos que a qualificação significa um pouco mais para classes (ela também é o lugar onde acontece algo chamado herança), mas, em geral, estas regras se aplicam a todos os nomes no Python.

Importações *versus* escopos

Nunca é possível acessar nomes definidos em outro arquivo de módulo, sem primeiro importar esse arquivo. Isto é, você nunca pode ver nomes em outro arquivo automaticamente, independente da estrutura das importações ou das chamadas de função existentes em seu programa.

Por exemplo, considere os dois módulos simples a seguir. O primeiro, *moda.py*, define uma variável global `X` apenas no código desse arquivo, junto com uma função que altera a variável global `X` desse arquivo:

```
X = 88                          # Meu X: global apenas para esse arquivo

def f():
    global X                    # Altera meu X.
    X = 99                      # Não pode ver nomes em outros módulos
```

O segundo módulo, *modb.py*, define sua própria variável global `X`, e importa e chama a função do primeiro módulo:

```
X = 11                          # Meu X: global apenas para esse arquivo

import moda                     # Obtém acesso aos nomes presentes em moda.
moda.f()                        # Configura moda.X e não meu X
print X, moda.X
```

Quando executado, `moda.f` altera o `X` em moda e não o `X` em `modb`. O escopo global de `moda.f` é sempre o arquivo que o envolve, independente do módulo a partir do qual ele é finalmente chamado:

```
% python modb.py
11 99
```

Em outras palavras, as operações de importação nunca fornecem visibilidade para cima no código de arquivos importados – não se pode ver nomes no arquivo que se está importando. Mais formalmente:

- As funções nunca podem ver nomes em outras funções, a não ser que elas sejam fisicamente envolventes.
- Código de módulo nunca pode ver nomes em outros módulos, a não ser que eles sejam explicitamente importados.

Esse comportamento faz parte da noção de *escopo léxico* – no Python, os escopos que cercam um trecho de código são completamente determinados pela posição física do código em seu arquivo. Os escopos nunca são influenciados pelas chamadas de função nem pelas importações de módulo.*

Aninhamento de espaço de nome

De certa forma, embora as importações não aninhem espaços de nome para cima, elas aninham para baixo. Usando caminhos de qualificação de atributo, é possível descer para módulos arbitrariamente aninhados e acessar seus atributos. Por exemplo, considere os três arquivos a seguir. *mod3.py* define um único nome global e um atributo por atribuição:

```
X = 3
```

mod2.py importa o primeiro e usa qualificação para acessar o atributo do módulo importado:

```
X = 2
import mod3

print X,                    # Meu X global
print mod3.X                # X de mod3
```

mod1.py importa o segundo e busca atributos do primeiro e do segundo arquivos:

```
X = 1
import mod2

print X,                        # Meu X global
print mod2.X,                   # X de mod2
print mod2.mod3.X               # X de mod3 aninhado
```

Na verdade, quando `mod1` importa `mod2` aqui, ele estabelece um aninhamento de espaço de nome de dois níveis. Usando o caminho de nomes `mod2.mod3.X`, ele desce para `mod3`, que está aninhado no arquivo importado `mod2`. O resultado é que `mod1` pode ver as variáveis `X` de todos os três arquivos e, por isso, tem acesso a todos os três escopos globais:

```
% python mod1.py
2 3
1 2 3
```

* Algumas linguagens agem de maneira diferente e fornecem escopo dinâmico, onde os escopos podem, na verdade, depender de chamadas em tempo de execução. Contudo, isso tende a tornar o código mais complicado, pois o significado de uma variável pode diferir com o passar do tempo.

Inversamente, mod3 não pode ver nomes em mod2 e este não pode ver nomes em mod1. Esse exemplo pode ser mais fácil de entender se você não pensar em termos de espaços de nome e escopos. Em vez disso, focalize os objetos envolvidos. Dentro de mod1, mod2 é apenas um nome que se refere a um objeto com atributos, alguns dos quais podem se referir a outros objetos com atributos (import é uma atribuição). Para caminhos como mod2.mod3.X, o Python simplesmente avalia da esquerda para a direita, buscando atributos de objetos pelo caminho.

Note que mod1 pode dizer import mod2 e depois mod2.mod3.X, mas não pode dizer import mod2.mod3 – essa sintaxe evoca algo chamado importações de pacote (diretório), descritas no próximo capítulo. As importações de pacote também criam aninhamento de espaço de nome de módulo, mas suas instruções de importação refletem árvores de diretório e não simples encadeamentos de importação.

RECARREGANDO MÓDULOS

Por padrão, o código de um módulo é executado apenas uma vez por processo. Para obrigar o código de um módulo a ser recarregado e novamente executado, você precisa pedir ao Python explicitamente para que faça isso, chamando a função interna reload. Nesta seção, exploraremos o uso de recarregamentos para tornar seus sistemas mais dinâmicos. Em resumo:

- As importações (tanto instruções import como from) carregam e executam o código de um módulo apenas na primeira vez que o módulo é importado em um processo.
- As importações posteriores usam o objeto módulo já carregado, sem recarregar nem executar novamente o código do arquivo.
- A função reload obriga o código de um módulo já carregado a ser recarregado e novamente executado. As atribuições no novo código do arquivo fazem alteração no local do objeto módulo existente.

Por que todo esse estardalhaço sobre o recarregamento de módulos? A função reload permite que partes de programas sejam alteradas sem interromper o programa inteiro. Com ela, os efeitos das alterações em componentes podem ser observados imediatamente. O recarregamento não ajuda em todas as situações, mas onde ajuda ele contribui para um ciclo de desenvolvimento muito mais curto. Por exemplo, imagine um programa de banco de dados que precisa se conectar com um servidor na inicialização. Como as alterações do programa podem ser testadas imediatamente após os recarregamentos, você precisa conectar-se apenas uma vez, enquanto depura.

Como o Python é interpretado (mais ou menos), ele já se livra das etapas de compilação/vinculação pelas quais você precisa passar para fazer um programa em C executar: os módulos são carregados dinamicamente, quando importados por um programa em execução. O recarregamento amplia isso, permitindo que você também altere partes de programas em execução sem parar. Devemos notar que, atualmente, reload só funciona em módulos escritos em Python; módulos de extensão C também podem ser carregados dinamicamente, mas eles não podem ser recarregados.

Fundamentos do recarregamento

Ao contrário das instruções import e from:

- reload é uma função interna no Python e não uma instrução.
- reload recebe um objeto módulo existente e não um nome.

Como reload espera um objeto, um módulo deve ter sido importado anteriormente com êxito, antes que você possa recarregá-lo. Na verdade, se a importação foi mal sucedida devido a um

erro de sintaxe ou outro, talvez você precise repeti-la antes de poder recarregar. Além disso, a sintaxe das instruções de importação e das chamadas de `reload` é diferente: os recarregamentos exigem parênteses, mas as importações, não. Veja um exemplo de recarregamento, a seguir:

```
import module                        # Importação inicial
...usa módulo.atributos...
...                                  # Agora, altera o arquivo de módulo.
...
reload(module)                       # Obtém exportações atualizadas.
...usa módulo.atributos...
```

Normalmente, você importa um módulo, depois altera seu código-fonte em um editor de textos e recarrega. Quando você chama `reload`, o Python lê novamente o código-fonte do arquivo de módulo e executa suas instruções de nível superior outra vez. Mas talvez o mais importante a saber sobre `reload` seja que ele altera um objeto módulo no local; essa função não exclui e recria o objeto módulo. Por isso, toda referência para um objeto módulo em qualquer parte de seu programa é automaticamente afetada por um recarregamento. Os detalhes:

`reload` executa o novo código de um arquivo de módulo no espaço de nome corrente do módulo. Executar novamente o código de um arquivo de módulo sobrescreve seu espaço de nome existente, em vez de excluí-lo e recriá-lo.

As atribuições de nível superior no arquivo substituem os nomes por novos valores. Por exemplo, executar novamente uma instrução `def` substitui a versão anterior da função no espaço de nome do módulo, reatribuindo o nome da função.

Os recarregamentos têm impacto em todos os clientes que usam `import` para buscar módulos. Como os clientes que usam `import` qualificam para buscar atributos, eles encontrarão novos valores no objeto módulo, após a execução de uma função `reload`.

Os recarregamentos têm impacto apenas nos futuros clientes de `from`. Os clientes que usavam `from` para buscar atributos no passado não serão afetados por uma função `reload`; eles ainda terão referências para os antigos objetos buscados antes do recarregamento.

Exemplo de recarregamento

Aqui está um exemplo mais concreto de `reload` em ação. No exemplo a seguir, alteramos e recarregamos um arquivo de módulo, sem interromper a sessão interativa do Python. Os recarregamentos também são usados em muitos outros cenários (veja o quadro "Por que isto é relevante: recarregamentos de módulo"), mas manteremos as coisas simples aqui, para ilustração. Primeiramente, vamos escrever um arquivo de módulo *changer.py* com o editor de textos de nossa escolha:

```
message = "First version"

def printer():
    print message
```

Esse módulo cria e exporta dois nomes – um vinculado a uma string e outro a uma função. Agora, inicie o interpretador do Python, importe o módulo e chame a função que ele exporta. A função imprime o valor da variável global message:

```
% python
>>> import changer
>>> changer.printer()
First version
>>>
```

Por que isto é relevante: recarregamentos de módulo

Além de permitirem que você recarregue (e, portanto, execute novamente) módulos no prompt interativo, os recarregamentos de módulo também são úteis em sistemas maiores, especialmente quando o custo de reiniciar um aplicativo inteiro é proibitivo. Por exemplo, os sistemas que precisam se conectar com servidores por meio de uma rede na inicialização são excelentes candidatos para recarregamentos dinâmicos.

Os recarregamentos também são úteis em trabalho com GUI (a ação de callback de um elemento de janela pode ser alterada enquanto a GUI permanece ativa) e quando o Python é usado como uma linguagem incorporada em um programa em C ou C++ (o programa envolvente pode solicitar um recarregamento do código Python que executa, sem precisar ser interrompido). Consulte o livro Programming Python para ver mais informações sobre recarregamento de callbacks de GUI e código Python incorporado.

Em geral, os recarregamentos permitem que os programas forneçam interfaces altamente dinâmicas. Por exemplo, o Python é freqüentemente usado como uma linguagem de *personalização* para sistemas maiores – os usuários podem personalizar produtos escrevendo trechos de código Python no local, sem ter de recompilar o produto inteiro (ou mesmo ter seu código-fonte). Nesses ambientes, o código Python já acrescenta um tempero dinâmico por si só.

Contudo, para serem ainda mais dinâmicos, tais sistemas podem recarregar automaticamente o código de personalização em Python sempre que ele é executado – desse modo, as alterações dos usuários são captadas enquanto o sistema está funcionando. Não há necessidade de parar e reiniciar cada vez que o código Python é modificado. Nem todos os sistemas exigem essa estratégia dinâmica, mas alguns exigem.

Em seguida, mantenha o interpretador ativo e edite o arquivo de módulo em outra janela:

```
...modifique changer.py sem parar o Python...
% vi changer.py
```

Aqui, altere a variável global `message`, assim como o miolo da função `printer`:

```
message = "After editing"

def printer():
    print 'reloaded:', message
```

Finalmente, volte para a janela do Python e recarregue o módulo para buscar o novo código que acabamos de alterar. Note que importar o módulo novamente não tem nenhum efeito; obtemos a mensagem original, mesmo que o arquivo tenha sido alterado. Precisamos chamar `reload` para obter a nova versão:

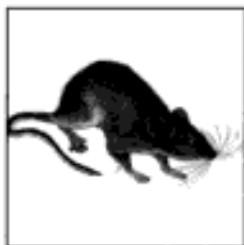
```
...de volta ao interpretador/programa Python...

>>> import changer
>>> changer.printer()        # Nenhum efeito: usa o módulo carregado
First version

>>> reload(changer)          # Obriga o novo código a ser carregado/executado
<module 'changer'>
>>> changer.printer()        # Executa a nova versão agora
reloaded: After editing
```

Note que, na verdade, `reload` retorna o objeto módulo para nós. Seu resultado normalmente é ignorado, mas como os resultados de expressão são impressos no prompt interativo, o Python mostra uma representação padrão `<módulo nome>`.

17

Pacotes de Módulo

Até aqui, quando importamos um módulo, carregamos arquivos. Isso representa a utilização normal de módulo e é o que você provavelmente usará para a maioria das importações que desenvolver no início de sua carreira no Python. A história da importação de módulo é um pouco mais rica do que sugerimos até agora.

As importações podem nomear um caminho de diretório, além de um nome de módulo. Quando fazem isso, elas são conhecidas como *importações de pacote* – diz-se que um diretório de código Python é um pacote. Esse é um recurso um tanto avançado, mas se mostra útil para organizar os arquivos em um sistema grande e tende a simplificar as configurações de caminho de pesquisa de módulo. Conforme veremos, às vezes as importações de pacote também são exigidas para solucionar ambigüidades, quando vários programas são instalados em uma única máquina.

FUNDAMENTOS DA IMPORTAÇÃO DE PACOTE

Aqui está como as importações de pacote funcionam. No lugar onde estivemos nomeando um arquivo simples, podemos, em vez disso, listar um caminho de nomes separados por pontos:

```
import dir1.dir2.mod
```

O mesmo vale para instruções `from`:

```
from dir1.dir2.mod import x
```

O caminho "com pontos" nessas instruções deve corresponder a um caminho pela hierarquia de diretórios em sua máquina, levando ao arquivo *mod.py* (ou outro tipo de arquivo). Isto é, existe o diretório `dir1`, que tem um subdiretório `dir2`, o qual contém um arquivo de módulo *mod.py* (ou outro sufixo).

Além disso, essas importações implicam que `dir1` reside dentro de algum diretório contêiner `dir0`, que é acessível no caminho de pesquisa de módulo do Python. Em outras palavras, as duas instruções de importação sugerem uma estrutura de diretório semelhante à seguinte (mostrada com separadores de barra invertida do DOS):

```
dir0\dir1\dir2\mod.py            # Ou mod.pyc,mod.so,...
```

O diretório contêiner `dir0` ainda precisa ser adicionado em seu caminho de pesquisa de módulo (a não ser que seja o diretório de base do arquivo de nível superior), exatamente como se `dir1` fosse um arquivo de módulo. De lá para baixo, as instruções de importação em seu script fornecem o caminho de diretório que leva explicitamente ao módulo.

Pacotes e configurações de caminho de pesquisa

Se você usar esse recurso, lembre-se de que os caminhos de diretório, em suas instruções de importação, só podem ser variáveis separadas por pontos. Você não pode usar qualquer sintaxe de caminho específica de uma plataforma em suas instruções de importação. Coisas como `C:\dir1`, `Meus Documentos.dir2` e `../dir1` não funcionam sintaticamente. Em vez disso, use sintaxe específica da plataforma em suas configurações de caminho de pesquisa de módulo, para nomear o diretório contêiner.

Por exemplo, no exemplo anterior, `dir0` – o nome de diretório que você adiciona em seu caminho de pesquisa de módulo – pode ser um caminho de diretório específico de uma plataforma, arbitrariamente longo, levando até `dir1`. Em vez de usar uma instrução inválida como esta:

```
import C:\mycode\dir1\dir2\mod        # Erro: sintaxe inválida
```

adicione `C:\mycode` em sua variável `PYTHONPATH` ou arquivos *.pth*, a menos que ele seja o diretório de base do programa, e escreva isto:

```
import dir1.dir2.mod
```

Na verdade, as entradas no caminho de pesquisa de módulo fornecem prefixos de caminho de diretório específicos da plataforma, os quais levam aos nomes mais à esquerda nas instruções de importação. As instruções de importação fornecem os finais do caminho de diretório de maneira neutra quanto a plataforma.*

Arquivos de pacote __init__.py

Se você optar por usar importações de pacote, há mais uma restrição que deve atender. Cada diretório nomeado dentro do caminho de uma instrução de importação de pacote também deve conter um arquivo chamado *__init__.py*, senão suas importações falharão. No exemplo que estávamos usando, `dir1` e `dir2` devem ambos conter um arquivo chamado *__init__.py*; o diretório contêiner `dir0` não exige esse arquivo, pois não é listado na instrução de importação em si. Mais formalmente, para uma estrutura de diretório como:

```
dir0\dir1\dir2\mod.py
```

e para uma instrução de importação da forma:

```
import dir1.dir2.mod
```

as seguintes regras se aplicam:

- `dir1` e `dir2` devem ambos conter um arquivo *__init__.py*.
- `dir0`, o contêiner, não exige um arquivo *__init__.py*; se estiver presente, ele simplesmente será ignorado.

* A sintaxe de caminho com pontos foi escolhida parcialmente pela neutralidade quanto a plataforma, mas também porque os caminhos nas instruções de importação tornam-se caminhos de objeto aninhado reais. Essa sintaxe também significa que você recebe mensagens de erro estranhas, se esquece de omitir o sufixo *.py* em suas instruções de importação: `import mod.py` deve ser uma importação de caminho de diretório – ela carrega *mod.py* e, em seguida, tenta carregar um *mod\py.py* e, em última análise, emite uma mensagem de erro potencialmente confusa.

- `dir0` deve ser listado no caminho de pesquisa de módulo (diretório de base, `PYTHONPATH` etc.) e não `dir0\dir1`.

O resultado é que a estrutura de diretório desse exemplo deve ser como a seguinte, com a endentação designando o aninhamento de diretório:

```
dir0\                    # Contêiner no caminho de pesquisa de módulo
    dir1\
        __init__.py
        dir2\
            __init__.py
            mod.py
```

Esses arquivos *__init__.py* contêm código Python, assim como os arquivos de módulo normais. Eles estão presentes parcialmente como uma declaração para o Python e podem ser completamente vazios. Como uma *declaração*, esses arquivos servem para evitar que diretórios com nome comum ocultem involuntariamente módulos reais que ocorram posteriormente no caminho de pesquisa de módulo. Caso contrário, o Python poderá escolher um diretório que nada tenha a ver com seu código, simplesmente porque ele aparece em um diretório anterior no caminho de pesquisa.

Em geral, esse arquivo serve como um gancho para ações em tempo de inicialização de pacote, para gerar um espaço de nome de módulo para um diretório e implementar o comportamento de instruções `from*` (isto é, `from ... import *`), quando usadas em importações de diretório:

Inicialização de pacotes

Na primeira vez que o Python importa por meio de um diretório, ele executa automaticamente todo o código presente no arquivo *__init__.py* do diretório. Por isso, esses arquivos são lugares naturais para se colocar código para inicializar o estado exigido pelos arquivos no pacote. Por exemplo, um pacote poderia usar seu arquivo de inicialização para criar os arquivos de dados exigidos, abrir conexões com bancos de dados etc. Normalmente, os arquivos *__init__.py* não se destinam a ter utilidade se executados diretamente. Eles são executados automaticamente durante as importações, na primeira vez que o Python passa por um diretório.

Inicialização de espaço de nome de módulo

No modelo de importação de pacotes, os caminhos de diretório presentes em seu script tornam-se caminhos de objeto aninhado reais, após a importação. Assim, no exemplo anterior, a expressão `dir1.dir2` funciona e retorna um objeto módulo cujo espaço de nome contém todos os nomes atribuídos pelo arquivo *__init__.py* de `dir2`. Tais arquivos fornecem um espaço de nome para módulos que não têm nenhum outro arquivo.

*Comportamento da instrução from**

Como um recurso avançado, você pode usar listas `__all__` em arquivos *__init__.py* para definir o que é exportado quando um diretório é importado com a forma de instrução `from*`. (Vamos conhecer `__all__` no Capítulo 18.) Em um arquivo *__init__.py*, a lista `__all__` é considerada a lista de nomes de sub-módulo que devem ser importados quando `from*` for usada no nome do pacote (diretório). Se `__all__` não for configurada, a instrução `from*` não carregará automaticamente os sub-módulos aninhados no diretório, mas, em vez disso, carregará apenas os nomes definidos pelas atribuições presentes no arquivo *__init__.py* do diretório, incluindo todos os sub-módulos explicitamente importados pelo código desse arquivo. Por exemplo, uma instrução `from submodule import X` no arquivo *__init__.py* de um diretório torna o nome `X` disponível no espaço de nome desse diretório.

Você também pode simplesmente deixar esses arquivos vazios, se suas funções estiverem além de suas necessidades. Contudo, eles devem existir realmente para que suas importações de diretório funcionem.

EXEMPLO DE IMPORTAÇÃO DE PACOTE

Vamos desenvolver o exemplo sobre o qual estivemos falando, para mostrar como os arquivos de inicialização e os caminhos funcionam. Os três arquivos a seguir são escritos em um diretório `dir1` e em seu subdiretório `dir2`:

```
#Arquivo: dir1\__init__.py
print 'dir1 init'
x = 1

#Arquivo: dir1\dir2\__init__.py
print 'dir2 init'
y = 2

#Arquivo: dir1\dir2\mod.py
print 'in mod.py'
z = 3
```

Aqui, `dir1` será um subdiretório do diretório em que estamos trabalhando (isto é, o diretório de base) ou um subdiretório de um diretório listado no caminho de pesquisa de módulo (tecnicamente, em `sys.path`). De qualquer modo, o contêiner de `dir1` não precisa de um arquivo *__init__.py*.

No que diz respeito a arquivos de módulo simples, as instruções de importação executam o arquivo de inicialização de cada diretório à medida que o Python desce no caminho, na primeira vez que um diretório é percorrido. Adicionamos instruções `print` para rastrear sua execução. Também como os arquivos de módulo, os diretórios já importados podem ser passados para `reload` para obrigar uma nova execução desse único item – `reload` aceita um nome de caminho com pontos, para recarregar diretórios e arquivos aninhados:

```
% python
>>> import dir1.dir2.mod           # As primeiras importações
                                     executam arquivos init.
dir1 init
dir2 init
in mod.py
>>> import dir1.dir2.mod           # As importações posteriores, não.
>>>
>>> reload(dir1)
dir1 init
<module 'dir1' from 'dir1\__init__.pyc'>
>>>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
```

Uma vez importado, o caminho em sua instrução `import` torna-se um *caminho de objeto aninhado* em seu script; `mod` é um objeto aninhado no objeto `dir2`, aninhado no objeto `dir1`:

```
>>> dir1
<module 'dir1' from 'dir1\__init__.pyc'>
>>> dir1.dir2
```

```
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from 'dir1\dir2\mod.pyc'>
```

Na verdade, cada nome de diretório no caminho torna-se uma variável atribuída a um objeto módulo, cujo espaço de nome é inicializado por todas as atribuições presentes no arquivo *__init__.py* desse diretório. `dir1.x` refere-se à variável `x` atribuída em **dir1***__init__.py*, exatamente como `mod.z` refere-se à variável `z` atribuída em *mod.py*:

```
>>> dir1.x
1
>>> dir1.dir2.y
2
>>> dir1.dir2.mod.z
3
```

from *versus* import com pacotes

As instruções `import` podem ser bastante inconvenientes para usar com pacotes, pois você freqüentemente precisa digitar novamente os caminhos em seu programa. No exemplo da seção anterior, você precisa digitar novamente o caminho completo de `dir1`, sempre que quer atingir `z`. Na verdade, obteremos erros aqui, se tentarmos acessar `dir` ou `mod` diretamente, neste ponto.

```
>>> dir2.mod
NameError: name 'dir2' is not defined
>>> mod.z
NameError: name 'mod' is not defined
```

Por isso, freqüentemente é mais conveniente usar a instrução `from` com pacotes, para evitar a digitação repetida de caminhos em cada acesso. Talvez o mais importante seja que, se você reestruturar sua árvore de diretório, a instrução `from` exigirá apenas uma atualização de caminho em seu código, enquanto a instrução `import` poderá exigir muitas. A extensão `import as`, discutida no próximo capítulo, também pode ajudar aqui, fornecendo um sinônimo mais curto para o caminho completo:

```
% python
>>> from dir1.dir2 import mod      # Escreve o caminho apenas aqui.
dir1 init
dir2 init
in mod.py
>>> mod.z                          # Não repete o caminho.
3
>>> from dir1.dir2.mod import z
>>> z
3
>>> import dir1.dir2.mod as mod    # Usa nome mais curto.
>>> mod.z
3
```

POR QUE USAR IMPORTAÇÕES DE PACOTE?

Se você é iniciante no Python, certifique-se de ter dominado os módulos simples, antes de progredir para os pacotes, pois eles são um recurso um tanto avançado da linguagem. Eles têm funções úteis, especialmente em programas maiores: eles tornam as importações mais

informativas, servem como uma ferramenta organizacional, simplificam seu caminho de pesquisa de módulo e podem solucionar ambigüidades.

Antes de tudo, como as importações de pacote fornecem algumas informações de diretório em arquivos de programa, elas tornam mais fácil localizar seus arquivos e servem como uma ferramenta organizacional. Sem os caminhos de pacote, você precisa contar com a consulta da pesquisa de módulo para localizar arquivos com mais freqüência. Além disso, se você organizar seus arquivos em subdiretórios de áreas funcionais, as importações de pacote tornarão mais evidente a função que um módulo desempenha e, portanto, tornará seu código mais legível. Por exemplo, uma importação normal de um arquivo em um diretório em algum lugar no caminho de pesquisa de módulo:

```
import utilities
```

transmite muito menos conhecimento do que uma importação que inclui informações de caminho:

```
import database.client.utilities
```

As importações de pacote também podem simplificar bastante suas configurações de caminho de pesquisa de `PYTHONPATH` ou de arquivo *.pth*. Na verdade, se você usar importações de pacote para todas as suas importações entre diretórios e torná-las relativas a um diretório-raiz comum, onde todo seu código Python é armazenado, só precisará de uma única entrada em seu caminho de pesquisa: a raiz comum.

A HISTÓRIA DOS TRÊS SISTEMAS

Contudo, a única vez em que as importações de pacote são realmente exigidas é na solução de ambigüidades que podem surgir quando vários programas são instalados em uma única máquina. Esse é um problema de instalação, mas também pode tornar-se uma preocupação na prática geral. Vamos recorrer a um cenário hipotético para ilustrar.

Suponha que um programador desenvolva um programa em Python que contenha um arquivo chamado *utilities.py* para código utilitário comum, e um arquivo de nível superior chamado *main.py* que os usuários executam para iniciar o programa. Por todo esse programa, seus arquivos contêm as instruções `import utilities` para carregar e usar o código comum. Quando esse programa é distribuído, ele chega como um único arquivo *tar* ou *zip*, contendo todos os seus arquivos. Quando ele é instalado, desempacota todos os seus arquivos em um único diretório chamado *system1*, na máquina de destino:

```
system1\
    utilities.py       # Funções utilitárias comuns, classes
    main.py            # Ative isto para iniciar o programa.
    other.py           # Importa utilitários para carregar minhas ferramentas
```

Agora, suponha que um segundo programador faça a mesma coisa: ele desenvolve um programa diferente com os arquivos *utilities.py* e *main.py*, e usa `import utilities` para carregar novamente o arquivo de código comum. Quando esse segundo sistema é buscado e instalado, seus arquivos são desempacotados em um novo diretório chamado `system2`, em algum lugar na máquina receptora, de modo que seus arquivos não sobrescrevem os arquivos de mesmo nome do primeiro sistema. Finalmente, os dois sistemas tornam-se tão populares que acabam sendo comumente instalados no mesmo computador:

```
system2\
    utilities.py       # Utilitários comuns
```

```
        main.py            # Ativa isto para executar.
        other.py           # Importa utilitários
```

Até aqui, não há nenhum problema: os dois sistemas podem coexistir ou executar na mesma máquina. Na verdade, nem mesmo precisamos configurar o caminho de pesquisa de módulo para usar esses programas – como o Python sempre pesquisa o diretório de base primeiro (isto é, o diretório que contém o arquivo de nível superior), as importações nos arquivos de qualquer um dos sistemas verão automaticamente todos os arquivos no diretório desse sistema. Por exemplo, se você clicar em *system1\main.py*, todas as importações pesquisarão `system1` primeiro. Analogamente, se você ativar *system2\main.py*, então `system2` será pesquisado primeiro, em vez do outro. Lembre-se de que as configurações de caminho de pesquisa de módulo só são necessárias para importar além dos limites de diretório.

Mas, agora, suponha que, após ter instalado esses dois programas em sua máquina, você decida que gostaria de usar código dos arquivos *utilities.py* de um dos dois em um sistema independente. Afinal, trata-se de código utilitário comum e, por natureza, o código Python quer ser reutilizado. Você deseja escrever o seguinte no código que está produzindo, em um terceiro diretório:

```
import utilities
utilities.func('spam')
```

para carregar um dos dois arquivos. Agora, o problema começa a se materializar. Para fazer isso funcionar, você terá que configurar o caminho de pesquisa de módulo para incluir os diretórios que contêm os arquivos *utilities.py*. Mas qual diretório você coloca primeiro no caminho – `system1` ou `system2`?

O problema é a natureza *linear* do caminho de pesquisa; ele é sempre percorrido da esquerda para a direita. Independente de quanto tempo você medite sobre esse dilema, sempre obterá *utilities.py* do diretório listado *primeiro* no caminho de pesquisa (o mais à esquerda). No estado em que se encontra, você nunca poderá importá-lo do outro diretório. Você poderia tentar alterar `sys.path` dentro de seu script, antes de cada operação de importação, mas isso é trabalho extra e altamente propenso a erro. Por padrão, você está encalhado.

Além disso, esse é o problema que os pacotes corrigem. Em vez de instalar programas como uma lista simples de arquivos em diretório independentes, empacote-os e os instale como *subdiretórios* sob uma raiz comum. Por exemplo, você poderia organizar todo o código desse exemplo como uma hierarquia de instalação semelhante à seguinte:

```
root\
    system1\
        __init__.py
        utilities.py
        main.py
        other.py
    system2\
        __init__.py
        utilities.py
        main.py
        other.py
system3\                        # Aqui ou em qualquer lugar
        __init__.py             # Seu novo código fica aqui
        myfile.py
```

Por que isto é relevante: pacotes de módulo

Agora que os pacotes são uma parte padrão do Python, é comum ver extensões maiores, de terceiros, distribuídas como um conjunto de diretórios de pacote, em vez de uma lista simples de módulos. O pacote de extensões para o Python win32all do Windows, por exemplo, foi um dos primeiros a aderir a isso. Muitos de seus módulos utilitários residem em pacotes importados com caminhos; por exemplo, para carregar ferramentas COM no lado do cliente:

```
from win32com.client import constants, Dispatch
```

essa linha busca nomes do módulo `client` do pacote `win32com` (um subdiretório). As importações de pacote também são freqüentes no código executado no Jython, a implementação do Python baseada em Java, pois as bibliotecas Java também são organizadas em uma hierarquia. Veremos mais sobre COM e Jython posteriormente, na Parte VII. Nas versões recentes do Python, as ferramentas de email e XML também são organizadas em subdiretórios empacotados na biblioteca padrão.

Agora, adicione apenas o diretório comum `root` em seu caminho de pesquisa. Se as importações de seu código são relativas a essa raiz comum, você pode importar *um ou outro* arquivo de utilitários do sistema com importações de pacote – o nome de diretório envolvente torna o caminho (e, portanto, a referência de módulo) exclusivo. Na verdade, você pode importar *os dois* arquivos de utilitários no mesmo módulo, contanto que use a instrução de importação e repita o caminho completo cada vez que referenciar módulos de utilitário:

```
import system1.utilities
import system2.utilities
system1.utilities.function('spam')
system2.utilities.function('eggs')
```

Note que foram adicionados arquivos *__init__.py* nos diretórios `system1` e `system2` para fazer isso funcionar, mas não na raiz: apenas diretórios listados dentro de instruções de importação exigem esses arquivos.

Tecnicamente, seu diretório `system3` não precisa estar sob `root` – apenas os pacotes de código a partir dos quais você importará. Entretanto, como você nunca sabe quando seus próprios módulos poderão ser úteis em outros programas, também pode colocá-los sob a raiz comum para evitar problemas de conflito de nomes semelhantes no futuro.

Além disso, note que as importações dos dois sistemas originais continuarão funcionando no estado em que se encontram e permanecerão inalteradas: como o diretório *de base* deles é pesquisado primeiro, a adição da raiz comum no caminho de pesquisa é irrelevante para o código em `system1` e em `system2`. Eles podem continuar dizendo apenas `import utilities` e encontrar seus próprios arquivos. Além disso, se você tomar o cuidado de desempacotar todos os sistemas Python sob a raiz comum dessa forma, a configuração de caminho se tornará simples: você só precisará adicionar a raiz comum uma vez.

18

Tópicos Avançados dos Módulos

A Parte V termina com um conjunto de tópicos mais avançados relacionados aos módulos, junto com o grupo padrão de problemas e exercícios. Exatamente como as funções, os módulos são mais eficientes quando suas interfaces são bem definidas; portanto, este capítulo também dá uma breve examinada nos conceitos do projeto de módulos. Alguns dos assuntos aqui, como o truque `__name__`, são muito utilizados, apesar da palavra "avançados" no título desde capítulo.

OCULTAÇÃO DE DADOS EM MÓDULOS

Conforme vimos, os módulos do Python exportam todos os nomes atribuídos no nível superior de seus arquivos. Não há nenhuma noção de declaração de quais nomes devem e não devem ser visíveis fora do módulo. Na verdade, não há nenhuma maneira de impedir que um cliente altere nomes dentro de um módulo, se ele quiser.

No Python, a ocultação de dados em módulos é uma convenção e não uma restrição sintática. Se quiser danificar um módulo jogando seus nomes fora, você pode, mas ainda não conhecemos nenhum programador que quisesse fazer isso. Alguns puristas não concordam com essa atitude liberal com relação à ocultação de dados e alegam que isso significa que o Python não consegue implementar o encapsulamento. No entanto, o encapsulamento no Python tem mais a ver com empacotamento do que com restrição.

Minimizando o dano de from*: _X e __all__

Como um caso especial, prefixar nomes com um único sublinhado (por exemplo, `_X`) evita que eles sejam copiados quando um cliente importa com uma instrução `from*`. Na verdade, isso se destina apenas a minimizar a poluição do espaço de nome. Como `from*` copia todos os nomes, você pode receber mais do que queria (incluindo nomes que sobrescrevem outros no importador). Mas os sublinhados não são declarações "privadas": você ainda pode ver e alterar esses nomes com outras formas de importação, como a instrução `import`.

Um módulo pode obter um efeito de ocultação semelhante à convenção de atribuição de nomes `_X`, atribuindo a variável `__all__` a uma lista de strings de nome de variável, no nível superior do módulo. Por exemplo:

```
__all__ = ["Error", "encode", "decode"]        # Exporta apenas estes.
```

Quando esse recurso é usado, a instrução `from*` só copiará os nomes incluídos na lista `__all__`. Na verdade, isso é o inverso da convenção `_X`: `__all__` contém os nomes a serem copiados, mas `_X` identifica os nomes a não serem copiados. O Python procura uma lista `__all__` primeiro no módulo. Se não houver nenhuma definida, a instrução `from*` copiará todos os nomes sem um único sublinhado no início.

A lista `__all__` também só tem significado para a forma de instrução `from*` e não é uma declaração de privacidade. Os escritores de módulo podem usar qualquer um dos dois truques para implementar módulos que se comportam bem quando usados com `from*`. Veja a discussão sobre listas `__all__` nos arquivos de pacote *__init__.py*, no Capítulo 17; lá, elas declaram submódulos a serem carregados para uma instrução `from*`.

ATIVANDO FUTUROS RECURSOS DA LINGUAGEM

As mudanças na linguagem que possivelmente podem arruinar código já existente no futuro são introduzidas gradualmente. Inicialmente, elas aparecem como extensões opcionais, as quais são desativadas por padrão. Para ativar essas extensões, use uma instrução de importação especial, desta forma:

```
from __future__ import nomedorecurso
```

Geralmente, essa instrução deve aparecer no início de um arquivo de módulo (possivelmente após uma docstring), pois ela ativa compilação de código especial, de acordo com o módulo. Também é possível submeter essa instrução no prompt interativo para experimentar futuras mudanças da linguagem; então, o recurso estará disponível para o restante da sessão interativa.

Por exemplo, tivemos que usar isso no Capítulo 14, para demonstrarmos as funções geradoras, as quais exigem uma palavra-chave que ainda não é ativada por padrão (elas usam o nome de recurso `generators`). Também usamos essa instrução para ativar a divisão real para números, no Capítulo 4.

MODOS DE UTILIZAÇÃO MISTOS: __name__ E __main__

Aqui está um truque especial relacionado aos módulos que permite a você importar um arquivo como módulo e executá-lo como um programa independente. Cada módulo tem um atributo interno chamado `__name__`, que o Python configura automaticamente, como segue:

- Se o arquivo estiver sendo executado como um arquivo de programa de nível superior, `__name__` será configurado como a string `"__main__"`, quando ele começar.
- Se o arquivo estiver sendo importado, em vez disso, `__name__` será configurado com o nome do módulo, conforme é conhecido por seus clientes.

A conclusão é que um módulo pode testar seu próprio atributo `__name__` para determinar se está sendo executado ou importado. Por exemplo, suponha que criemos o arquivo de módulo a seguir, chamado *runme.py*, para exportar uma única função, chamada `tester`:

```
def tester():
    print "It's Christmas in Heaven..."

if __name__ == '__main__':              # Somente quando executa
    tester()                            # Não quando importado
```

Esse módulo define uma função para os clientes importarem e usarem, como sempre:

```
% python
>>> import runme
>>> runme.tester()
It's Christmas in Heaven'...
```

Mas o módulo também inclui código no final, que é configurado para chamar a função quando esse arquivo é executado como um programa:

```
% python runme.py
It's Christmas in Heaven...
```

Talvez o lugar mais comum em que você verá o teste `__name__` aplicado será em código de *auto-teste*: você pode empacotar código que testa as exportações de um módulo no próprio módulo, envolvendo-o em um teste de `__name__` no final. Desse modo, você pode usar o arquivo nos clientes, importando-o, e testar sua lógica executando-o a partir do shell do sistema ou de outros esquemas de execução. O Capítulo 26 discutirá outras opções comumente usadas para testar código Python.

Outra função comum do truque de `__name__` é na escrita de arquivos cuja funcionalidade pode ser usada como um utilitário de linha de comando *e* como uma biblioteca de ferramentas. Por exemplo, suponha que você escreva um script localizador de arquivo em Python; você pode usufruir mais de seu código se empacotá-lo em funções e adicionar um teste de `__name__` no arquivo para chamar essas funções automaticamente, quando o arquivo for executado de forma independente. Desse modo, o código do script torna-se reutilizável em outros programas.

ALTERANDO O CAMINHO DE PESQUISA DE MÓDULO

No Capítulo 15, mencionamos que o caminho de pesquisa de módulo é uma lista de diretórios inicializada a partir da variável de ambiente `PYTHONPATH` e, possivelmente, de arquivos de caminho *.pth*. O que não mostramos até agora é como um programa em Python pode mudar o caminho de pesquisa, alterando uma lista interna chamada `sys.path` (o atributo `path` no módulo interno `sys`). A lista `sys.path` é inicializada na partida do sistema, mas, depois disso, você pode excluir, anexar e reconfigurar seus componentes como quiser:

```
>>> import sys
>>> sys.path
['', 'D:\\PP2ECD-Partial\\Examples', 'C:\\Python22', ...more deleted...]
>>> sys.path = [r'd:\temp']                 # Altera o caminho de pesquisa de
>>> sys.path.append('c:\\lp2e\\examples')   # módulo apenas para esse processo.
>> sys.path
['d:\temp', 'c:\\lp2e\\examples']

>>> import string
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named string
```

Você pode usar isso para configurar um caminho de pesquisa dinamicamente dentro de um programa em Python. Cuidado: se você excluir um diretório importante do caminho, poderá perder o acesso a utilitários fundamentais. No último comando do exemplo, não temos mais acesso ao módulo string, pois excluímos do caminho o diretório da biblioteca de código-fonte Python. Lembre-se também de que essas configurações só permanecem durante a sessão ou programa em Python que as fez; elas não são mantidas depois que o Python sai.

A INSTRUÇÃO IMPORT COMO EXTENSÃO

As instruções `import` e `from` foram estendidas para permitir que um módulo receba nomes diferentes em seu script:

```
import longmodulename as name
```

é equivalente a:

```
import longmodulename
name = longmodulename
del longmodulename                    # Não mantém o nome original.
```

Após a instrução `import`, você pode (e, na verdade, deve) usar o nome que aparece depois de `as` para referir-se ao módulo. Isso também funciona em uma instrução `from`:

```
from module import longname as name
```

para atribuir um nome diferente para o nome do arquivo em seu script. Essa extensão é normalmente usada para fornecer sinônimos curtos para nomes mais longos e para evitar conflitos de nome quando você já está usando um nome em seu script, que de outra forma seria sobrescrito por uma instrução de importação normal. Isso também é útil para fornecer um nome simples e curto para um caminho de diretório inteiro, ao se usar o recurso de importação de pacotes descrito no Capítulo 17.

CONCEITOS DE DESIGN DE MÓDULOS

Assim como as funções, os módulos apresentam compromissos de design: decidir quais funções entram em cada módulo, os mecanismos de comunicação entre os módulos etc. Aqui estão algumas idéias gerais que se tornarão mais claras quando você começar a escrever sistemas maiores em Python:

Você está sempre em um módulo no Python. Não há como escrever código que não fique em algum módulo. Na verdade, o código digitado no prompt interativo fica em um módulo interno chamado `__main__`; os únicos detalhes exclusivos do prompt interativo são o código que é executado e descartado imediatamente e os resultados das expressões que são impressos.

Minimize o acoplamento de módulo: variáveis globais. Assim como as funções, os módulos funcionam melhor se são escritos como caixas fechadas. Como regra geral, eles devem ser o mais independentes possível dos nomes globais em outros módulos.

Maximize a coesão do módulo: objetivo unificado. Você pode minimizar os acoplamentos de um módulo maximizando sua coesão. Se todos os componentes de um módulo compartilham seu propósito geral, é menos provável que você dependa de nomes externos.

Raramente os módulos devem alterar variáveis de outros módulos. É perfeitamente válido usar variáveis globais definidas em outro módulo (que é como os clientes importam serviços), mas alterá-las em outro módulo freqüentemente é um sintoma de problema de design. Existem exceções, é claro, mas você deve tentar comunicar os resultados por meio de dispositivos como valores de retorno de função, e não por alterações entre módulos. Caso contrário, os valores de suas variáveis globais se tornarão dependentes da ordem de atribuições remotas arbitrárias.

Como um resumo, a Tabela 18-1 esboça o ambiente em que os módulos operam. Os módulos contêm variáveis, funções, classes e outros módulos (se forem importados). As funções têm suas próprias variáveis locais. Você vai conhecer as classes – outro objeto que fica dentro dos módulos – no Capítulo 19.

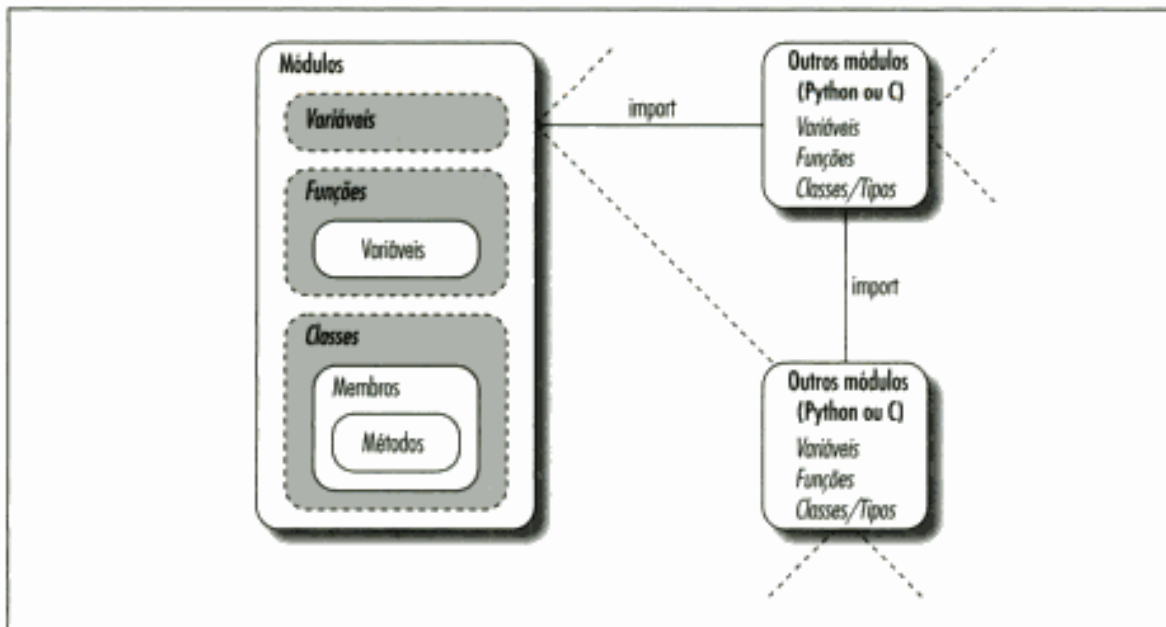



Figura 18-1 Ambiente do módulo.

Os módulos são objetos: meta-programas

Como os módulos expõem a maioria de suas propriedades interessantes como atributos internos é fácil escrever programas que gerenciam outros programas. Normalmente, chamamos esses programas gerenciadores de *meta-programas*, pois eles funcionam em cima de outros sistemas. Isso também é referido como *introspecção*, pois os programas podem ver e processar componentes internos do objeto. A introspecção é um recurso avançado, mas pode ser útil para construir ferramentas de programação.

Por exemplo, para chegar a um atributo chamado `name` em um módulo chamado `M`, podemos usar qualificação ou indexar o dicionário de atributos do módulo, exposto no atributo interno `__dict__`. Além disso, o Python também exporta a lista de todos os módulos carregados como o dicionário `sys.modules` (isto é, o atributo `modules` do módulo `sys`) e fornece uma função interna chamada `getattr` que nos permite buscar atributos de seus nomes de string (é como escrever `object.attr`, mas `attr` é uma string em tempo de execução). Por isso, todas as expressões a seguir atingem o mesmo atributo e objeto:

```
M.name                    # Qualifica objeto.
M.__dict__['name']        # Indexa manualmente o dicionário de espaço de nome.
sys.modules['M'].name     # Indexa manualmente a tabela de módulos carregados.
getattr(M, 'name')        # Chama a função de busca interna.
```

Expondo os componentes internos do modo, dessa forma, o Python o ajuda a construir programas sobre programas.* Por exemplo, aqui está um módulo chamado *mydir.py* que coloca essas idéias para trabalhar, implementando uma versão personalizada da função interna `dir`. Ele define e exporta uma função chamada `listing`, que recebe como argumento um objeto módulo e imprime uma listagem formatada do espaço de nome do módulo:

* Note que, como uma função pode acessar seu módulo envolvente, passando pela tabela `sys.modules`, dessa forma, é possível simular o efeito da instrução global que você conheceu no Capítulo 13. Por exemplo, o efeito de `global X; X=0` pode ser simulado escrevendo-se, dentro de uma função: `import sys; glob=sys.modules[__name__]; glob.X=0` (se bem que com muito mais digitação). Lembre-se de que cada módulo recebe um atributo `__name__` gratuitamente; ele é visível como um nome global dentro de funções que estão dentro de um módulo. O truque oferece outra maneira de alterar tanto variáveis locais como globais de mesmo nome, dentro de uma função.

```
# Um módulo que lista os espaços de nome de outros módulos

verbose = 1

def listing(module):
    if verbose:
        print "-"*30
        print "name:", module.__name__, "file:", module.__file__
        print "-"*30

    count = 0
    for attr in module.__dict__.keys():  # Percorre o espaço de nome.
        print "%02d) %s" % (count, attr),
        if attr[0:2] == "__":
            print "<built-in name>"      # Pula __file__ etc.
        else:
            print getattr(module, attr)  # O mesmo que .__dict__[attr]
        count = count+1

    if verbose:
        print "-"*30
        print module.__name__, "has %d names" % count
        print "-"*30

    if __name__ == "__main__":
    import mydir
    listing(mydir)                       # Código de auto-teste: lista myself
```

Também fornecemos lógica de auto-teste no final desse módulo, o qual, de forma narcisista, importa e lista a si mesmo. Aqui está o tipo de saída produzida:

```
C:\python> python mydir.py
------------------------------
name: mydir file: mydir.py
------------------------------
00) __file__ <built-in name>
01) __name__ <built-in name>
02) listing <function listing at 885450>
03) __doc__ <built-in name>
04) __builtins__ <built-in name>
05) verbose 1
------------------------------
mydir has 6 names
------------------------------
```

Vamos encontrar `getattr` e seus parentes novamente. O ponto a notar aqui é que `mydir` é um programa que permite a você percorrer outros programas. Como o Python expõe seus componentes internos, você pode processar objetos genericamente.*

PROBLEMAS DOS MÓDULOS

Aqui está a coleção usual de casos-limite, os quais tornam a vida interessante para os iniciantes. Alguns são tão obscuros que foi difícil sugerir exemplos, mas a maioria ilustra algo importante sobre o Python.

* Ferramentas como `mydir.listing` podem ser previamente carregadas no espaço de nome interativo, importando-as no arquivo referenciado pela variável de ambiente `PYTHONSTARTUP`. Como o código presente no arquivo de inicialização é executado no espaço de nome interativo (módulo `__main__`), as importações de ferramentas comuns no arquivo de inicialização economizam alguma digitação de sua parte. Consulte o Apêndice A para ver mais detalhes.

Importando módulos pela string de nome

O nome do módulo em uma instrução `import` ou `from` é um nome de variável incorporado ao código. Contudo, às vezes seu programa terá o nome de um módulo importado como uma string em tempo de execução (por exemplo, se um usuário seleciona um nome de módulo dentro de uma GUI). Infelizmente, você não pode usar instruções de importação diretamente para carregar um módulo, dado seu nome como uma string – o Python espera uma variável aqui e não uma string. Por exemplo:

```
>>> import "string"
   File "<stdin>", line1
   import "string"
                 ^
SyntaxError: invalid syntax
```

Colocar a string em um nome de variável também não funcionaria:

```
x = "string"
import x
```

Aqui, o Python tentará importar um arquivo *x.py* e não o módulo `string`.

Para contornar esse problema, você precisa usar ferramentas especiais para carregar módulos dinamicamente de uma string existente em tempo de execução. A estratégia mais geral é construir uma instrução `import` como uma string de código Python e passá-la para a instrução `exec` executar:

```
>>> modname = "string"
>>> exec "import " + modname          # Executa uma string de código.
>>> string                            # Importada neste espaço de nome
<module 'string'>
```

A instrução `exec` (e sua prima para expressões, a função `eval`) compila uma string de código e a passa para o interpretador do Python para ser executada. No Python, o compilador de código de byte está disponível em tempo de execução; portanto, você pode escrever programas que constróem e executam outros programas, como esse. Por padrão, a instrução `exec` executa o código no escopo corrente, mas você pode ser mais específico, passando dicionários de espaço de nome opcionais.

O único inconveniente da instrução `exec` é que ela precisa compilar a instrução `import` sempre que é executada. Se ela for executada muitas vezes, seu código poderá ser mais rápido se usar a função interna `__import__` para carregar a partir de uma string de nome. O efeito é semelhante, mas `__import__` retorna o objeto módulo; portanto, para mantê-lo, atribua um nome a ele aqui:

```
>>> modname = "string"
>>> string = __import__(modname)
>>> string
<module 'string'>
```

A instrução from copia nomes, mas não vincula

A instrução `from` é, na realidade, uma atribuição para nomes no escopo do importador – uma operação de cópia de nome e não um alias de nome. As implicações disso são as mesmas de todas as atribuições no Python, mas são sutis, especialmente dado que o código que compartilha objetos fica em arquivos diferentes. Por exemplo, suponha que definamos o seguinte módulo (*nested1.py*):

```
X = 99
def printer(): print X
```

Se importarmos seus dois nomes usando `from` em outro módulo (*nested2.py*), obteremos cópias desses nomes e não vínculos para eles. Alterar um nome no importador reconfigura apenas o vínculo da versão local desse nome e não o nome em *nested1.py*:

```
from nested1 import X, printer        # Copia nomes.
X = 88                                # Altera apenas meu "X"!
printer()                             # o X de nested1 ainda é 99

% python nested2.py
99
```

Se você usar `import` para obter o módulo inteiro e atribuir a um nome qualificado, alterará o nome em *nested1.py*. A qualificação direciona o Python para um nome no objeto módulo, em vez de um nome no importador (*nested3.py*):

```
import nested1                        # Obtém o módulo como um todo.
nested1.X = 88                        # Tudo bem: altera X de nested1
nested1.printer()

% python nested3.py
88
```

A ordem da instrução é importante no código de nível superior

Quando um módulo é importado pela primeira vez (ou recarregado), o Python executa suas instruções, uma por uma, do início ao fim do arquivo. Isso tem algumas implicações sutis relacionadas às referências que ocorrerão adiante, que valem a pena sublinhar aqui:

- O código no nível superior de um arquivo de módulo (não aninhado em uma função) é executado assim que o Python o atinge, durante uma importação. Por isso, ele não pode referenciar nomes atribuídos mais abaixo no arquivo.
- O código dentro do miolo de uma função não é executado até que a função seja chamada. Como os nomes presentes em uma função não são solucionados até que a função seja realmente executada, normalmente eles podem referenciar nomes em qualquer parte do arquivo.

Geralmente, as referências que ocorrerão adiante são uma preocupação no código do módulo de nível superior que é executado imediatamente; as funções podem referenciar nomes arbitrariamente. Aqui está um exemplo que ilustra a referência que ocorrerá adiante:

```
func1()                   # Erro: "func1" ainda não foi atribuída

def func1():
    print func2           # Tudo bem: "func2" é pesquisada posteriormente

func1()                   # Erro: "func2" ainda não foi atribuída

def func2():
    return "Hello"

func1()                   # Tudo bem: "func1" e "func2" foram atribuídas
```

Quando esse arquivo é importado (ou executado como um programa independente), o Python executa suas instruções do início ao fim. A primeira chamada para `func1` falha porque sua instrução `def` ainda não foi executada. A chamada para `func2` dentro de `func1` funciona, desde que a instrução `def` de `func2` tenha sido atingida no momento em que `func1` é chamada (ela ainda não

foi, quando a segunda chamada de `func1` de nível superior é executada). A última chamada para `func1`, no final do arquivo, funciona, pois `func1` e `func2` foram ambas atribuídas.

Misturar instruções `def` com código de nível superior não é algo apenas difícil de ler, mas também é dependente da ordem das instruções. Como regra geral, se você precisa misturar código imediato com instruções `def`, coloque as instruções `def` no início do arquivo e o código de nível superior no final. Desse modo, suas funções estarão definidas e atribuídas quando o código que as utiliza for executado.

Importações "from" recursivas podem não funcionar

Como as importações executam as instruções de um arquivo do início ao fim, às vezes você precisa tomar cuidado ao usar módulos que importam um ao outro (às vezes chamadas de *importações recursivas*). Como as instruções de um módulo não foram ainda todas executadas, quando ele importa outro módulo, alguns de seus nomes podem ainda não existir. Se você usar `import` para buscar um módulo como um todo, isso pode ter importância ou não; os nomes do módulo não serão acessados até que você utilize qualificação, posteriormente, para buscar seus valores. Mas se você usar `from` para buscar nomes específicos, só terá acesso aos nomes já atribuídos.

Por exemplo, tomemos os módulos `recur1` e `recur2`, a seguir. `recur1` atribui um nome `X` e, depois, importa `recur2`, antes de atribuir o nome `Y`. Nesse ponto, `recur2` pode buscar `recur1` como um todo, com uma instrução `import` (ele já existe na tabela de módulos interna do Python), mas só pode ver o nome `X` se usar `from`; o nome `Y` abaixo da instrução `import` em `recur1` ainda não existe, de modo que você obtém um erro:

```
#Arquivo: recur1.py
X = 1
import recur2              # Executa recur2 agora, se ele ainda não existe.
Y = 2


#Arquivo: recur2.py
from recur1 import X       # Tudo bem: "X" já foi atribuído
from recur1 import Y       # Erro: "Y" ainda não foi atribuído

>>> import recur1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "recur1.py", line 2, in ?
    import recur2
  File "recur2.py", line 2, in?
    from recur1 import Y     # Erro: "Y" ainda não foi atribuído
ImportError: cannot import name Y
```

O Python evita a nova execução das instruções de `recur1`, quando elas são importadas recursivamente de `recur2` (senão, as importações fariam o script entrar em um loop infinito), mas o espaço de nome de `recur1` está incompleto quando é importado por `recur2`.

Não use `from` em importações recursivas...mesmo! O Python não ficará travado em um ciclo, mas seus programas, mais uma vez, ficarão dependentes da ordem das instruções nos módulos. Existem duas maneiras de evitar esse problema:

- Normalmente, você pode eliminar ciclos de importação como esse com um projeto cuidadoso. Maximizar a coesão e minimizar o acoplamento são bons primeiros passos.
- Se você não conseguir quebrar os ciclos completamente, adie o acesso ao nome do módulo usando `import` e qualificação (em vez de `from`) ou executando suas instruções `from`

dentro de funções (em vez de executar no nível superior do módulo) ou próximo do fim do seu arquivo, para retardar sua execução.

A função reload pode não ter impacto nas importações com from

A instrução `from` é a fonte de todos os tipos de problemas no Python. Aqui está outro: como a instrução `from` copia (atribui) nomes quando executada, não há nenhum vínculo de volta para o módulo de onde os nomes vieram. Os nomes importados com `from` tornam-se simplesmente referências para objetos, os quais foram referenciados pelos mesmos nomes no importador, quando a instrução `from` foi executada.

Por causa desse comportamento, recarregar o importado não tem nenhum efeito sobre os clientes que utilizam `from`. Os nomes do cliente ainda referenciam os objetos originais buscados com `from`, mesmo que os nomes no módulo original tenham sido reconfigurados:

```
from module import X    # X pode não refletir nenhum recarregamento de módulo!
...
reload(module)          # Altera o módulo, mas não meus nomes
X                       # Ainda referencia o objeto antigo
```

Não faça isso dessa maneira. Para tornar os recarregamentos mais eficientes, use `import` e qualificação de nome, em vez de `from`. Como as qualificações sempre voltam para o módulo, elas encontrarão os novos vínculos de nomes de módulo, após o recarregamento:

```
import module                   # Obtém o módulo e não os nomes.
...
reload(module)                  # Altera o módulo no local.
module.X                        # Obtém o X corrente: reflete os
                                  recarregamentos de módulo
```

reload, from e testes interativos

O Capítulo 3 alertou os leitores de que normalmente é melhor não executar programas com importações e recarregamentos, por causa das complexidades envolvidas. As coisas ficam ainda piores com `from`. Os iniciantes em Python freqüentemente encontram este problema: após abrirem um arquivo de módulo em uma janela do editor de textos, eles executam uma sessão interativa para carregar e testar seus módulos com `from`:

```
from module import function
function(1, 2, 3)
```

Após encontrar um erro, eles voltam para a janela de edição, fazem uma alteração e tentam recarregar desta maneira:

```
reload(module)
```

Só que isso não funciona – a instrução `from` atribuiu o nome `function` e não `module`. Para referir-se ao módulo em uma função `reload`, você precisa primeiro carregá-lo com uma instrução `import`, pelo menos uma vez:

```
import module
reload(module)
function(1, 2, 3)
```

Só que isso também não funciona – a função `reload` atualiza o objeto módulo, mas nomes, como `function`, copiados do módulo no passado ainda referem-se aos objetos antigos (neste caso, a versão original da função). Para realmente obter a nova função, chame-a com `module.function`, após a função `reload`, ou execute a instrução `from` novamente:

```
import module
reload(module)
from module import function
function(1, 2, 3)
```

Agora, a nova versão da função finalmente é executada. Mas existem problemas inerentes ao uso de `reload` com `from`. Você não apenas tem de lembrar-se de recarregar após as importações, como também precisa lembrar-se de executar novamente suas instruções `from`, após os recarregamentos. Isso é complexo o suficiente até para confundir um especialista de vez em quando.

Você não deve esperar que `reload` e `from` funcionem bem em conjunto. Melhor ainda, não as combine de jeito nenhum – use `reload` com `import` ou ative os programas de outras maneiras, conforme sugerido no Capítulo 3 (por exemplo, use a opção Edit/Runscript no IDLE, cliques em ícone de arquivo ou linhas de comando de sistema).

reload não é aplicado transitivamente

Quando você recarrega um módulo, o Python só recarrega o arquivo desse módulo em particular. Ele não recarrega automaticamente os módulos que o arquivo que está sendo recarregado importa. Por exemplo, se você recarrega algum módulo `A` e `A` importa os módulos `B` e `C`, o recarregamento só se aplica a `A` e não a `B` e `C`. As instruções dentro de `A` que importam `B` e `C` são executadas novamente, durante o recarregamento, mas apenas buscarão os objetos módulo `B` e `C` já carregados (supondo que eles não foram importados antes). Em código real, aqui está o arquivo *A.py*:

```
import B                    # Não recarregado quando A é recarregado
import C                    # Apenas uma importação de um módulo já carregado
```

```
% python
>>> . . .
>>> reload(A)
```

Não dependa de recarregamentos de módulo transitivos. Use várias chamadas de `reload` para atualizar subcomponentes de forma independente. Se desejar, você pode projetar seus sistemas para recarregarem seus subcomponentes automaticamente, adicionando chamadas de `reload` em módulos ascendentes, como `A`.

Melhor ainda, você poderia escrever uma ferramenta geral para fazer recarregamentos transitivos automaticamente, percorrendo o módulo `__dict__s` (veja a seção "Os módulos são objetos: meta-programas", anteriormente neste capítulo) e verificando o componente `type()` de cada item (veja o Capítulo 7) para encontrar módulos aninhados para recarregar recursivamente. Tal função utilitária poderia chamar a si mesma, recursivamente, para navegar arbitrariamente em encadeamentos de dependência de importação formados.

O módulo *reloadall.py*, listado a seguir, tem uma função `reload_all` que recarrega um módulo automaticamente, todo módulo que esse módulo importa e assim por diante, até o final dos encadeamentos de importação. Ela usa um dicionário para monitorar os módulos já recarregados, recursividade para percorrer os encadeamentos de importação e o módulo `types` da biblioteca padrão (apresentado no final do Capítulo 7), que simplesmente predefine o resultado de `type()` para tipos incorporados.

Para usar esse utilitário, importe sua função `reload_all` e passe a ela o nome de um módulo já carregado, exatamente como na função interna `reload`. Quando o arquivo é executado independentemente, seu código de auto-teste testa a si mesmo – ele precisa importar a si mesmo,

pois seu próprio nome não é definido no arquivo sem uma importação. O incitamos a estudar e experimentar este exemplo por conta própria:

```
import types

def status(module):
    print 'reloading', module.__name__

def transitive_reload(module, visited):
    if not visited.has_key(module):        # Captura ciclos, dups.
       status(module)                      # Recarrega este módulo
       reload(module)                      # e visita os filhos.
       visited[module] = None
       for attrobj in module.__dict__.values():   # Para todos os atributos
           if type(attrobj) == types.ModuleType:  # Recursividade se
                                                  for módulo
               transitive_reload(attrobj, visited)

def reload_all(*args):
    visited = {}
    for arg in args:
       if type(arg) == types.ModuleType:
           transitive_reload(arg, visited)

if __name__ == '__main__':
    import reloadall               # Código de teste: recarrega a mim mesmo
    reload_all(reloadall)          # Deve recarregar isto, types
```

EXERCÍCIOS DA PARTE V

1. *Fundamentos, importação*. Escreva um programa que conte linhas e caracteres em um arquivo (semelhante ao espírito de "wc" no Unix). Com seu editor de textos, desenvolva um módulo em Python chamado *mymod.py*, que exporte três nomes de nível superior:
   - Uma função `countLines(name)` que leia um arquivo de entrada e conte o número de linhas presentes (dica: `arquivo.readlines()` faz a maior parte do trabalho para você e `len` faz o resto)
   - Uma função `countChars(name)` que leia um arquivo de entrada e conte o número de caracteres presentes (dica: `arquivo.read()` retorna uma única string)
   - Uma função `test(name)` que chame as duas funções de contagem com determinado nome de arquivo de entrada. Geralmente, esse nome de arquivo pode ser passado, incorporado no código, inserido com `raw_input` ou extraído de uma linha de comando por meio da lista `sys.argv`. Por enquanto, suponha que ele recebe um argumento de função.

   Todas as três funções de `mymod` devem esperar que uma string de nome de arquivo seja passada. Se você digitar mais de duas ou três linhas por função, está trabalhando muito – use as dicas listadas anteriormente!

   Em seguida, teste seu módulo interativamente, usando importação e qualificação de nome para buscar suas exportações. Sua variável `PYTHONPATH` precisa incluir o diretório onde você criou `mymod.py`? Tente executar seu módulo nele mesmo: por exemplo, `test("mymod.py")`. Note que `test` abre o arquivo duas vezes. Se você estiver se sentindo ambicioso, pode aprimorar isso, passando um objeto arquivo aberto para as duas funções de contagem (dica: `arquivo.seek(0)` é um retrocesso no arquivo).

2. `from/from*`. Teste interativamente seu módulo `mymod` do Exercício 1, usando `from` para carregar as exportações diretamente, primeiro pelo nome e depois usando a variante `from*` para buscar tudo.
3. `__main__`. Adicione em seu módulo `mymod` uma linha que chame a função `test` automaticamente, apenas quando o módulo for executado como um script e não quando ele for importado. A linha adicionada provavelmente testará o valor de `__name__` para a string `"__main__"`, conforme mostrado neste capítulo. Tente executar seu módulo a partir da linha de comando do sistema. Em seguida, importe o módulo e teste suas funções interativamente. Ele ainda funciona nos dois modos?
4. *Importações aninhadas*. Escreva um segundo módulo, `myclient.py`, que importe `mymod` e teste suas funções. Execute `myclient` a partir da linha de comando do sistema. Se `myclient` usar `from` para buscar a partir de `mymod`, as funções de `mymod` estarão acessíveis no nível superior de `myclient`? E se, em vez disso, ele importar com `import`? Tente desenvolver as duas variações em `myclient` e testar interativamente, importando `myclient` e inspecionando seu atributo `__dict__`.
5. *Importações de pacote*. Importe seu arquivo a partir de um pacote. Crie um subdiretório chamado `mypkg`, aninhado em um diretório em seu caminho de pesquisa de importação de módulo, mova o arquivo de módulo `mymod.py` que você criou no Exercício 1 ou 3 para o novo diretório e tente importá-lo com uma importação de pacote da forma: `import mypkg.mymod`.

   Para fazer isso funcionar, você precisará adicionar um arquivo *__init__.py* no diretório para o qual seu módulo foi movido, mas isso deve funcionar em todas as principais plataformas Python (isso é parte do motivo pelo qual o Python usa "." como separador de caminho). O diretório de pacotes que você criar pode ser simplesmente um subdiretório daquele em que está trabalhando. Se for, ele será encontrado por meio do componente de diretório de base do caminho de pesquisa e você não terá que configurar seu caminho. Adicione algum código em seu arquivo *__init__.py* e veja se ele é executado em cada importação.
6. *Recarregamento*. Experimente os recarregamentos de módulo: realize os testes do exemplo `changer.py` do Capítulo 16, alterando repetidamente a mensagem e/ou comportamento da função chamada, sem interromper o interpretador do Python. Dependendo de seu sistema, você poderá editar `changer` em outra janela ou suspender o interpretador do Python e editar na mesma janela (no Unix, a combinação de teclas Ctrl-Z normalmente suspende o processo corrente e um comando `fg` posterior o retoma).
7. *Importações circulares*.* Na seção sobre problemas da importação recursiva, importar `recur1` gerava um erro. Mas se você reiniciar o Python e importar `recur2` interativamente, o erro não ocorre: teste e veja você mesmo. Por que você acha que isso funciona para importar `recur2`, mas não para `recur1`? (Dica: o Python armazena módulos novos na tabela interna `sys.modules` (um dicionário), antes de executar seus códigos. As importações posteriores buscam o módulo dessa tabela primeiro, esteja o módulo "completo" ou não.) Agora, tente executar `recur1` como um arquivo de script de nível superior: `% python recur1.py`. Você obtém o mesmo erro que ocorre quando `recur1` é importado interativamente? Por quê? (Dica: quando os módulos são executados como programas, eles não são importados; portanto, esse caso tem o mesmo efeito que importar `recur2` interativamente; `recur2` é o primeiro módulo importado.) O que acontece quando você executa `recur2` como um script?

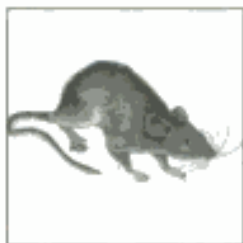
* Note que as importações circulares são extremamente raras na prática. Na verdade, este autor nunca escreveu nem se deparou com uma importação circular em uma década de desenvolvimento em Python. Por outro lado, se você consegue entender porque esse é um problema em potencial, então sabe muito sobre a semântica de importação do Python.

VI

Classes e POO

Na Parte VI, estudaremos os fundamentos da *programação orientada a objetos* (POO), assim como o código que você escreve para usar POO no Python – a instrução `class`. Conforme você vai ver, a POO é uma opção no Python, mas uma opção muito boa: nenhuma outra construção na linguagem suporta reutilização de código no grau que a instrução `class` suporta. Especialmente em programas maiores, a noção da POO de programação pela *personalização* é um paradigma poderoso para se aplicar e pode diminuir substancialmente o tempo de desenvolvimento, quando bem utilizada.

19

POO: o Panorama Geral

Até aqui, neste livro, estivemos usando o termo "objeto" genericamente. Na realidade, o código escrito até este ponto foi *baseado em objetos* – passamos objetos por todos os lados, os utilizamos em expressões, chamamos seus métodos etc. Contudo, para o código ser qualificado como verdadeiramente *orientado a objetos* (OO), geralmente os objetos também precisam participar de algo chamado hierarquia de herança.

Este capítulo inicia a exploração da *classe* do Python – um dispositivo usado para implementar novos tipos de objetos na linguagem. A classe é a principal ferramenta de *programação orientada a objetos* (POO) do Python; portanto, também vamos examinar os fundamentos da POO nesta parte do livro. No Python, as classes são criadas com uma nova instrução: `class`. Conforme veremos, os objetos definidos com classes podem ser muito parecidos com os tipos internos que examinamos anteriormente no livro. Eles também suportam herança – um mecanismo de personalização e reutilização de código, acima e além de tudo que vimos até agora.

Uma nota antecipada: a POO do Python é totalmente opcional e, para começo de conversa, você não precisa usar classes. Na verdade, você pode fazer muito trabalho com construções mais simples, como as funções, ou mesmo com código de script de nível superior simples. Mas as classes são uma das ferramentas mais úteis fornecidas pelo Python e mostraremos o motivo aqui. Elas também são empregadas em ferramentas populares do Python, como a API de GUI Tkinter; portanto, a maioria dos programadores de Python normalmente achará útil pelo menos um conhecimento operacional dos fundamentos das classes.

POR QUE USAR CLASSES?

Lembra-se de quando falamos que os programas fazem coisas com material? Em termos simples, as classes são apenas uma maneira de definir novos tipos de material que refletem objetos reais no domínio de seu programa. Suponha, por exemplo, que tenhamos decidido implementar aquele robô pizzaiolo hipotético que usamos no Capítulo 12. Se o implementarmos usando classes, poderemos modelar mais de sua estrutura real e de seus relacionamentos com o mundo:

Herança

Os robôs pizzaiolos são uma espécie de robô e, assim, possuem as propriedades robóticas comuns. Em termos de POO, dizemos que eles herdam propriedades da categoria geral de todos os robôs. Essas propriedades comuns precisam ser implementadas apenas uma vez para o caso geral, e reutilizadas por todos os tipos de robôs que possamos construir no futuro.

Composição

Os robôs pizzaiolos são, na realidade, uma coleção de componentes que trabalham em conjunto como uma equipe. Por exemplo, para que nosso robô tenha sucesso, ele precisaria de braços para enrolar a massa, motores para manusear o forno e assim por diante. No jargão da POO, nosso robô é um exemplo de composição: ele contém outros objetos que executa para cumprir suas ordens. Cada componente poderia ser desenvolvido como uma classe definindo seu próprio comportamento e seus relacionamentos.

As idéias gerais da POO, como herança e composição, são empregadas em qualquer aplicativo que possa ser decomposto em um conjunto de objetos. Por exemplo, nos sistemas de GUI típicos, as interfaces são escritas como coleções de elementos de janela – botões, rótulos etc. – todos os que são desenhados quando seus contêineres são desenhados (*composição*). Além disso, podemos escrever nossos próprios elementos de janela personalizados – botões com fontes exclusivas, rótulos com novos esquemas de cor e coisas assim – que são versões especializadas de dispositivos de interface mais gerais (*herança*).

A partir de uma perspectiva de programação mais concreta, as classes representam uma unidade de programa do Python, exatamente como as funções e os módulos. Elas são outro compartimento do empacotamento de lógica e dados. Na verdade, as classes também definem um novo espaço de nome, exatamente como os módulos. Mas comparadas com outras unidades de programa que já vimos, as classes têm três distinções fundamentais que as tornam mais úteis quando se trata de construir novos objetos:

Instâncias múltiplas

A grosso modo, as classes são fábricas para gerar um ou mais objetos. Sempre que chamamos uma classe, geramos um novo objeto, com um espaço de nome distinto. Cada objeto gerado a partir de uma classe tem acesso aos atributos da classe e recebe seu próprio espaço de nome para dados, que varia de acordo com o objeto.

Personalização por meio de herança

As classes também suportam a noção de herança da POO. Elas são estendidas pela redefinição de seus atributos fora da classe em si. Em geral, as classes podem construir hierarquias de espaços de nome, as quais definem os nomes a serem usados pelos objetos criados a partir das classes da hierarquia.

Sobrecarga de operador

Fornecendo métodos de protocolo especiais, as classes podem definir objetos que respondem às operações que vimos nos tipos internos. Por exemplo, os objetos feitos com classes podem ser fracionados, concatenados, indexados etc. O Python fornece ganchos que as classes podem usar para interceptar e implementar qualquer operação de tipo interno.

POO A 30.000 PÉS DE ALTURA

Antes de mostrarmos o que tudo isso significa em termos de código, gostaríamos de dizer aqui algumas palavras sobre as idéias gerais existentes por trás da POO. Se você nunca fez nada orientado a objetos em sua vida, algumas palavras que usaremos neste capítulo podem parecer

um pouco desconcertantes à primeira vista. Além disso, a motivação para usar essas palavras pode ser indefinível, até você ter tido uma chance de estudar as maneiras como os programadores as aplicam em sistemas maiores. A POO é tanto uma experiência quanto tecnologia.

Pesquisa de herança de atributo

A boa nova é que a POO é muito mais simples de entender e usar no Python do que em outras linguagens, como C++ ou Java. Como uma linguagem de script tipada dinamicamente, o Python elimina grande parte da confusão sintática e da complexidade que obscurece a POO em outras ferramentas. Na verdade, a maior parte da POO, no Python, se resume a esta expressão:

```
objeto.atributo
```

Estivemos usando isso por todo o livro, para acessar atributos de módulo, chamar métodos de objetos etc. Quando escrevemos isso para um objeto derivado de uma instrução `class`, a expressão provoca uma *pesquisa* no Python – ele pesquisa uma árvore de objetos vinculados, em busca da primeira aparição do **`atributo`** que puder encontrar. Na verdade, quando classes estão envolvidas, a expressão anterior em Python é traduzida para o seguinte, em linguagem natural:

```
Encontre a primeira ocorrência de atributo, procurando em objeto e em todas
as classes acima dele, de baixo para cima e da esquerda para a direita.
```

Em outras palavras, as buscas de atributo são simplesmente pesquisas de árvore. Chamamos esse procedimento de pesquisa de *herança*, pois, em uma árvore, os objetos mais abaixo herdam atributos ligados aos objetos mais acima, apenas porque a pesquisa de atributo ocorre de baixo para cima na árvore. De certo modo, a pesquisa automática realizada pela herança significa que os objetos vinculados em uma árvore são a união de todos os atributos definidos em todas as suas árvores ascendentes, até o topo da árvore.

No Python, isso é tudo muito literal: realmente construímos árvores de objetos vinculados com código e o Python realmente sobe nessa árvore em tempo de execução, procurando atributos, sempre que escrevemos `objeto.atributo`. Para tornar isso mais concreto, a Figura 19-1 esboça um exemplo de uma dessas árvores.

Nessa figura, existe uma árvore de cinco objetos rotulados com variáveis, todos os quais têm atributos vinculados, prontos para serem pesquisados. Mais especificamente, essa árvore vincula três *objetos classe* (as formas ovais: `C1, C2, C3`) e dois *objetos instância* (os retângulos: `I1, I2`), em uma árvore de pesquisa de herança. No modelo de objeto do Python, as classes e as instâncias que você gera a partir delas são dois tipos de objeto distintos:

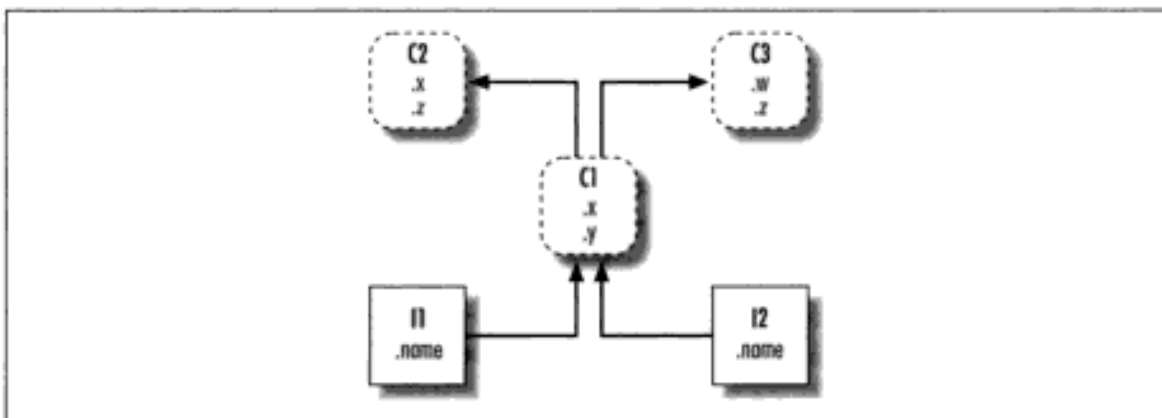



Figura 19-1 Uma árvore de classes.

Classes

Servem como fábricas de instância. Seus atributos fornecem comportamento – dados e funções – que é herdado por todas as instâncias geradas a partir delas (por exemplo, uma função para calcular salário de funcionários a partir do ordenado e das horas).

Instâncias

Representam os itens concretos no domínio de um programa. Seus atributos registram dados que variam de acordo com o objeto específico (por exemplo, o número da seguridade social de um funcionário).

Em termos de árvores de pesquisa, uma instância herda atributos de sua classe e uma classe herda atributos de todas as classes acima dela na árvore.

Na Figura 19-1, podemos categorizar ainda mais as formas ovais, pela sua posição relativa na árvore. Normalmente, chamamos as classes que estão mais acima na árvore (como `C2` e `C3`) de *superclasses*; as classes que estão mais abaixo (como `C1`) são conhecidas como *subclasses*.* Esses termos referem-se às posições relativas na árvore e às funções. Por causa da pesquisa de herança, as superclasses fornecem comportamento compartilhado por todas as suas subclasses. Como a pesquisa é de baixo para cima, as subclasses podem anular comportamento definido em suas superclasses, redefinindo os nomes da superclasse mais abaixo na árvore.

Como essas últimas palavras são realmente o ponto crucial da questão da personalização de software na POO, vamos expandir esse conceito. Suponha que tenhamos construído a árvore da Figura 19-1 e, depois, tenhamos escrito o seguinte:

```
I2.w
```

Agora mesmo, estamos usando herança aqui. Como essa é uma daquelas expressões `objeto.atributo`, ela provoca uma pesquisa na árvore da Figura 19-1. O Python procurará o atributo `w`, pesquisando em `I2` e acima. Especificamente, ela pesquisará os objetos vinculados nesta ordem:

```
I2, C1, C2, C3
```

e parará no primeiro `w` vinculado que encontrar (ou gerará um erro, caso ele não possa ser encontrado). Para essa expressão, `w` não seria encontrado até chegar a `C3`, como uma última etapa, pois ele só aparece nesse objeto. Em outras palavras, `I2.w` é solucionada como `C3.w`, graças à pesquisa automática. Na terminologia da POO, `I2` "herda" o atributo `w` de `C3`. Outras referências de atributo acabarão seguindo caminhos diferentes na árvore. Por exemplo:

- `I1.x e I2.x` encontram `x` em `C1` e param, pois `C1` está mais abaixo do que `C2`.
- `I1.y e I2.y` encontram `y` em `C1`, pois esse é o único lugar em que `y` aparece.
- `I1.z e I2.z` encontram `z` em `C2`, pois `C2` está mais à esquerda do que `C3`.
- `I2.name` encontra `name` em `I2`, sem subir na árvore.

Em última análise, as duas instâncias herdam quatro atributos de suas classes: `w`, `x`, `y` e `z`. Acompanhe essas pesquisas pela árvore na Figura 19-1 para entender como a herança faz sua pesquisa no Python. A primeira da lista acima talvez seja a mais importante a notar – como `C1` redefine o atributo `x` mais abaixo na árvore, ela efetivamente *substitui* a versão acima, em `C2`. Conforme você vai ver em breve, tais redefinições são o centro da personalização de software na POO.

* Em outra literatura, ocasionalmente você também pode ver os termos *classe de base* e *classe derivada* para descrever superclasses e subclasses, respectivamente.

Todas essas classes e objetos instância que colocamos nessas árvores são apenas pacotes de nomes, conhecidos como *espaços de nome* – lugares onde podemos vincular atributos. Se isso se parece com os módulos, deve mesmo parecer. A única diferença importante aqui é que os objetos nas árvores de classe também têm vínculos com outros objetos espaços de nome, pesquisados automaticamente.

Desenvolvendo árvores de classe

Embora estejamos falando de forma abstrata aqui, existe um código tangível por trás de todas essas idéias. Construímos essas árvores e seus objetos com instruções `class` e chamadas de classe, as quais conheceremos com mais detalhes. Mas, em resumo:

- Cada instrução `class` gera um novo objeto classe.
- Sempre que uma classe é chamada, ela gera um novo objeto instância.
- As instâncias são vinculadas automaticamente à classe a partir da qual são criadas.
- As classes são vinculadas às suas superclasses, sendo listadas entre parênteses em uma linha de cabeçalho da instrução `class`. A ordem da esquerda para a direita fornece a disposição na árvore.

Para construirmos a árvore da Figura 19-1, por exemplo, executaríamos código Python da seguinte forma (omitimos os detalhes das instruções `class` aqui):

```
class C2: ...               # Produz objetos classe (formas ovais).
class C3: ...
class C1(C2, C3): ...        # Vínculos para superclasses

I1 = C1()                    # Produz objetos instância (retângulos).
I2 = C1()                    # Vinculado às suas classes
```

Aqui, construímos os três objetos classe executando três instruções `class` e produzimos os dois objetos instância chamando uma classe duas vezes, como se fosse uma função. As instâncias lembram da classe a partir de onde foram produzidas e a classe `C1` lembra de suas superclasses listadas.

Tecnicamente, esse exemplo está usando algo chamado *herança múltipla*, que significa simplesmente que uma classe tem mais de uma superclasse acima dela na árvore de classes. No Python, a disposição da esquerda para a direita das superclasses listadas entre parênteses em uma instrução `class` (como as de `C1`, aqui) fornece a ordem na qual as superclasses são pesquisadas, se houver mais de uma.

Devido à maneira como a herança pesquisa, o objeto em que você anexa um atributo torna-se fundamental – ele determina o escopo do nome. Os atributos anexados nas instâncias pertencem apenas a uma única instância, mas os atributos anexados às classes são compartilhados por todas as suas subclasses e instâncias. Posteriormente, estudaremos com profundidade o código que pendura atributos nesses objetos. Conforme descobriremos:

- Os atributos normalmente são anexados às classes por meio de atribuições feitas dentro de instruções `class`.
- Os atributos são normalmente anexados às instâncias por meio de atribuições a um argumento especial, chamado `self`, passado para funções dentro das classes.

Por exemplo, as classes fornecem comportamento para suas instâncias com funções criadas pelo desenvolvimento de instruções `def` dentro de instruções `class`. Como essas instruções

dos de métodos de sobrecarga de operador. Esses métodos são herdados, como sempre, em árvores de classe e, para distingui-los, têm sublinhados duplos no início e no final de seus nomes. Eles são executados automaticamente pelo Python, quando objetos aparecem em expressões, e são principalmente uma alternativa para o uso de chamadas de método simples. Eles também são opcionais: se forem omitidos, a operação não será suportada.

Por exemplo, para implementar interseção de conjuntos, uma classe poderia fornecer um método chamado `intersect` ou sobrecarregar o operador de expressão `&` para executar a lógica exigida, escrevendo um método chamado `__and__`. Como o esquema do operador faz as instâncias se parecerem e se comportarem de modo mais parecido com os tipos internos, ele permite que algumas classes forneçam uma interface consistente e natural, e sejam compatíveis com código que espera um tipo interno.

A POO está relacionada à reutilização de código

Isso, junto com alguns detalhes da sintaxe, é a maior parte da história da POO no Python. Há um pouco mais sobre POO, no Python, do que a herança. Por exemplo, a sobrecarga de operadores é muito mais geral do que o descrito até aqui – as classes também podem fornecer a implementação de indexação, buscas de atributo, impressão e muito mais. De modo geral, contudo, a POO está relacionada à pesquisa de atributos em árvores.

Então, por que estaríamos interessados em construir e pesquisar árvores de objetos? Embora seja necessária alguma experiência para ver como, quando bem usadas, as classes suportam *reutilização* de código de maneiras que outros componentes de programa no Python não conseguem. Com classes, desenvolvemos *personalizando* software já existente, em vez de alterar código existente no local ou começar desde o início a cada novo projeto.

Em um nível básico, as classes são apenas pacotes de funções e outros nomes, exatamente como um módulo. Entretanto, a pesquisa de herança de atributo automática que obtemos com as classes suporta personalização de software muito além dos módulos e funções. Além disso, as classes fornecem uma estrutura natural para código que localiza lógica e nomes, ajudando assim na depuração.

Por exemplo, como os métodos são simplesmente funções com um primeiro argumento especial, podemos reproduzir parte de seu comportamento passando objetos manualmente para serem processados por funções simples. Contudo, a participação de métodos na *herança* de classe nos permite personalizar software existente de forma natural, desenvolvendo subclasses com novas definições de método, em vez de alterar código existente no local. Não há nenhum conceito assim nos módulos e nas funções.

Aqui está um exemplo: suponha que você seja incumbido de implementar um aplicativo de banco de dados de funcionários. Como programador de POO em Python, você poderia começar desenvolvendo uma superclasse geral que definisse o comportamento padrão comum a todos os tipos de funcionários em sua empresa:

```
class Employee:                         # Superclasse geral
    def computeSalary(self): ...        # Comportamento comum ou padrão
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

Uma vez desenvolvido esse comportamento geral, você pode especializá-lo para cada tipo de funcionário específico que seja diferente da norma. Você escreve subclasses que personalizam apenas o pouco comportamento que difere nos tipos de funcionários; o comportamento restan-

te será herdado da classe mais geral. Por exemplo, se os engenheiros têm uma regra de cálculo de salário exclusiva (talvez não sejam as horas trabalhadas), substitua apenas esse único método em uma subclasse:

```
class Engineer(Employee):                  # Subclasse especializada
     def computeSalary(self): ...          # Algo personalizado aqui
```

Como a versão de `computeSalary` aqui está mais abaixo na árvore de classes, ela substituirá (anulará) a versão geral existente em `Employee`. Crie instâncias do tipo de classe de funcionário ao qual um funcionário real pertence, para obter o comportamento correto. Note que podemos criar instâncias de qualquer classe em uma árvore e não apenas das que estão na parte inferior – a classe a partir da qual você cria uma instância determina o nível no qual a pesquisa de atributos começará:

```
bob = Employee()                           # Comportamento padrão
mel = Engineer()                           # Calculadora de salário personalizada
```

Em última análise, esses dois objetos instância poderiam acabar incorporados em um objeto contêiner maior (por exemplo, uma lista ou uma instância de outra classe) que representasse um departamento ou empresa, usando a idéia da *composição*, mencionada no início deste capítulo. Quando solicitarmos os salários, posteriormente, eles serão calculados de acordo com a classe a partir da qual o objeto foi criado, devido à pesquisa de herança – um outro caso da idéia de *polimorfismo* para funções, apresentada no Capítulo 12:*

```
company = [bob, mel]                       # Um objeto composto
for emp in company:
    print emp.computeSalary()              # Executa a versão deste objeto
```

Polimorfismo quer dizer que o significado de uma operação depende do objeto em que se está operando. Aqui, o método `computeSalary` é localizado pela herança em cada objeto, antes que ele seja chamado. Em outras aplicações, o polimorfismo também poderia ser usado para ocultar (isto é, *encapsular*) diferenças de interface. Por exemplo, um programa que processa fluxos de dados poderia ser desenvolvido para esperar objetos com métodos de entrada e saída, sem se preocupar com o que esses métodos realmente fazem:

```
def processor(reader, converter, writer):
    while 1:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Passando instâncias de subclasses que especializam as interfaces de método `read` e `write` exigidas para várias fontes de dados, a função processor pode ser reutilizada para qualquer fonte de dados que precisemos usar, tanto agora como no futuro:

```
class Reader:
    def read(self): ...                    # Comportamento e ferramentas padrão
    def other(self): ...
class FileReader(Reader):
    def read(self)                         # Lê um arquivo local
class SocketReader(Reader):
    def read(self): ...                    # Lê um soquete de rede
```

* Note que a lista de empresas nesse exemplo poderia ser armazenada em um arquivo usando a função `pickle` nos objetos do Python, apresentada posteriormente neste livro, para produzir um banco de dados de funcionários persistente.

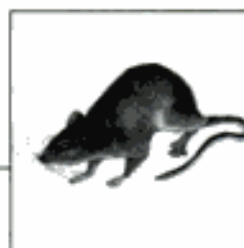
```
...
processor(FileReader(...), Converter, FileWriter(...))
processor(SocketReader(...), Converter, TapeWriter(...))
processor(FtpReader(...), Converter, XmlWriter(...))
```

Além disso, a implementação interna desses métodos `read` e `write` pode ser alterada sem afetar o código que os utiliza, como esse. Na verdade, a própria função `processor` poderia ser uma classe, para permitir que a lógica de conversão de converter fosse preenchida por herança, e incorporar leitores e gravadores por composição (veremos como fazer isso posteriormente nesta parte do livro).

Quando estiver acostumado a programar por meio de *personalização de software*, dessa maneira, você verá que, ao escrever um novo programa, grande parte de seu trabalho já estará feita – sua tarefa torna-se principalmente misturar as superclasses existentes que já implementam o comportamento exigido por seu programa. Por exemplo, `Employee` e as classes leitora e gravadora desses exemplos já podem ter sido escritas por outras pessoas, para uso em um programa completamente diferente. Se assim for, você obterá todo o código delas "gratuitamente".

Na verdade, em muitos domínios de aplicação, você pode buscar ou adquirir coleções de superclasses, conhecidas como modelos, que implementam tarefas de programação comuns como classes prontas para serem misturadas em seus aplicativos. Esses modelos podem fornecer interfaces de banco de dados, protocolos de teste, kits de ferramentas de GUI etc. Com os modelos, você freqüentemente desenvolve apenas uma subclasse que preenche um ou dois métodos esperados. As classes do modelo que estão mais acima na árvore fazem a maior parte do trabalho para você. A programação nesse mundo da POO é apenas uma questão de combinar e especializar código já depurado, escrevendo suas próprias subclasses.

É claro que demora certo tempo para aprender a promover as classes para obter tal utopia da POO. Na prática, o trabalho orientado a objetos também envolve bastante projeto para se perceber completamente as vantagens da reutilização de código das classes. Com esse objetivo, os programadores começaram a catalogar estruturas de POO comuns, conhecidas como *padrões de projeto*, para ajudar nos problemas de projeto. O código que você escreve para utilizar POO no Python é tão simples que, por si só, não criará nenhum obstáculo adicional em suas jornadas na POO. Para ver porque, você terá que passar para o Capítulo 20.

20

Fundamentos do Desenvolvimento de Classes

Agora que já falamos sobre a POO de forma abstrata, vamos passar aos detalhes de como isso se transforma em código real. Neste capítulo e no próximo, completaremos os detalhes da sintaxe existente por trás do modelo de classe no Python.

Se você nunca foi exposto à POO no passado, as classes podem ser um tanto complicadas, se encaradas de uma só vez. Para tornar mais fácil absorver o desenvolvimento de classes, começaremos nosso exame detalhado da POO vendo primeiro as classes em ação neste capítulo. Expandiremos os detalhes apresentados aqui em capítulos posteriores desta parte do livro, mas em sua forma básica as classes do Python são fáceis de entender.

As classes têm três distinções principais. Em um nível básico, elas são principalmente apenas *espaços de nome*, muito parecidos com os módulos estudados na Parte V. Mas, ao contrário dos módulos, as classes também têm suporte para gerar vários objetos, herança de espaço de nome e sobrecarga de operadores. Vamos começar nosso passeio pela instrução `class` explorando cada uma dessas três distinções por sua vez.

AS CLASSES GERAM VÁRIOS OBJETOS INSTÂNCIA

Para entender como a idéia dos vários objetos funciona, você precisa primeiro compreender que existem dois tipos de objetos no modelo de POO do Python – objetos *classe* e objetos *instância*. Os objetos classe fornecem comportamento padrão e servem como fábricas de objetos instância. Os objetos instância são os objetos reais processados por seus programas; cada um é um espaço de nome por si mesmo, mas herda (isto é, tem acesso automático aos) nomes da classe a partir da qual foi criado. Os objetos classe são provenientes de instruções e as instâncias, de chamadas; sempre que chama uma classe, você obtém uma nova instância dessa classe.

Esse conceito de geração de objetos é muito diferente de qualquer uma das outras construções de programa que estudamos até agora neste livro. Na verdade, as classes são fábricas para produzir

muitas instâncias. Em contraste, existe apenas uma cópia de cada módulo importado (na verdade, essa é uma razão de precisarmos chamar a função `reload` para atualizar o único objeto módulo).

A seguir, resumiremos os fundamentos da POO do Python. De certa forma, as classes são semelhantes às instruções `def` e aos módulos, mas podem ser muito diferentes do que você utilizou em outras linguagens.

Os objetos classe fornecem comportamento padrão

A instrução class cria um objeto classe e atribui um nome a ele. Exatamente como a instrução de função `def`, a instrução `class` do Python é *executável*. Quando obtida e executada, ela gera um novo objeto classe e atribui um nome a ele no cabeçalho da classe. Também como a instrução `def`, as instruções `class` normalmente são executadas quando o arquivo em que estão escritas é importado pela primeira vez.

As atribuições dentro de instruções class produzem atributos de classe. Assim como os arquivos de módulo, as *atribuições* dentro de uma instrução `class` geram *atributos* em um objeto classe. Após a execução de uma instrução `class`, os atributos da classe são acessados por qualificação de nome: `objeto.nome`.

Os atributos da classe fornecem o estado e o comportamento do objeto. Os atributos de um objeto classe gravam as informações de estado e o comportamento, para serem compartilhados por todas as instâncias criadas a partir da classe. As instruções de função `def` aninhadas dentro de uma instrução `class` geram *métodos*, os quais processam instâncias.

Os objetos instância são itens concretos

Chamar um objeto classe como uma função produz um novo objeto instância. Sempre que uma classe é chamada, ela cria e retorna um novo objeto instância. As instâncias representam itens concretos no domínio de seu programa.

Cada objeto instância herda atributos da classe e recebe seu próprio espaço de nome. Os objetos instância criados a partir de classes são novos espaços de nome; eles começam vazios, mas herdam atributos que ficam no objeto classe a partir do qual foram gerados.

As atribuições aos atributos de self em métodos produzem atributos por instância. Dentro de funções de método de classe, o primeiro argumento (chamado `self`, por convenção) referencia o objeto instância que está sendo processado. As atribuições aos atributos de `self` criam ou alteram dados na instância e não na classe.

Um primeiro exemplo

Vamos ver um exemplo real para mostrar como essas idéias funcionam na prática. Para começar, vamos definir uma classe chamada `FirstClass`, executando uma instrução `class` do Python interativamente:

```
>>> class FirstClass:                       # Define um objeto classe.
...     def setdata(self, value):           # Define métodos de classe.
...         self.data = value               # Self é a instância.
...     def display(self):
...         print self.data                 # self.data: por instância
```

Estamos trabalhando interativamente aqui, mas normalmente uma instrução assim seria executada quando o arquivo de módulo em que ela está escrita fosse importado. Assim como as funções, sua classe não existirá até que o Python obtenha e execute essa instrução.

Assim como todas as instruções compostas, `class` começa com uma linha de cabeçalho que lista o nome da classe, seguida de um miolo com uma ou mais instruções aninhadas e (normalmente) endentadas. Aqui, as instruções aninhadas são instruções `def`; elas definem funções que implementam o comportamento que a classe deve exportar. Conforme aprendemos, na verdade, a instrução `def` é uma atribuição; aqui, ela atribui aos nomes `setdata` e `display` no escopo da instrução `class` e, portanto, gera atributos vinculados à classe: `FirstClass.setdata` e `FirstClass.display`.

Normalmente, as funções dentro de uma classe são chamadas de *métodos*; elas são instruções `def` normais, mas o primeiro argumento recebe automaticamente um objeto instância implícito, quando chamadas – o sujeito de uma chamada. Precisamos de duas instâncias para vermos como isso acontece:

```
>>> x = FirstClass()                # Produz duas instâncias.
>>> y = FirstClass()                # Cada uma é um novo espaço de nome.
```

Chamando-se a classe dessa maneira (observe os parênteses), ela gera objetos instância, os quais são apenas espaços de nome que têm acesso aos atributos de sua classe. Falando corretamente, neste ponto temos três objetos – duas instâncias e uma classe. Na realidade, temos três espaços de nome vinculados, conforme esboçado na Figura 20-1. Em termos de POO, dizemos que `x` "é" uma `FirstClass`, assim como `y`.

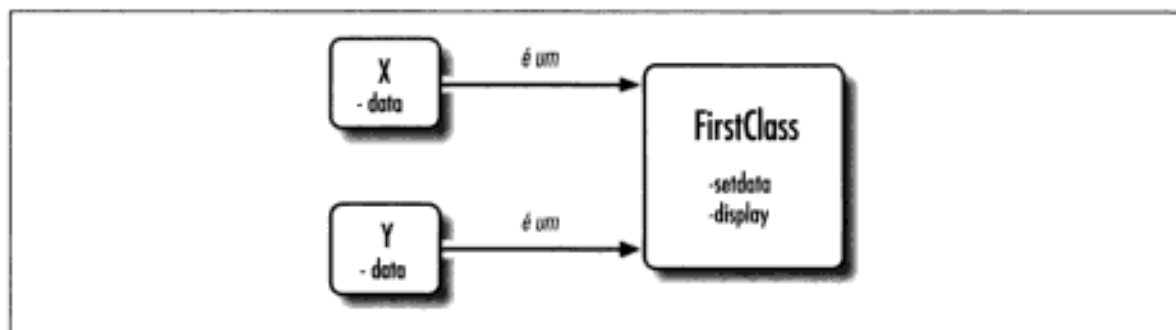



Figura 20-1 As classes e instâncias são objetos espaço de nome vinculados.

As duas instâncias começam vazias, mas têm vínculos de volta para a classe a partir da qual foram geradas. Se qualificarmos uma instância com o nome de um atributo residente no objeto classe, o Python buscará o nome da classe por meio de pesquisa de herança (a não ser que ele também resida na instância):

```
>>> x.setdata("King Arthur")        # Chama métodos: self é x
>>> y.setdata(3.14159)              # Executa: FirstClass.setdata(y, 3.14159)
```

Nem `x` nem `y` tem sua própria função `setdata`; em vez disso, o Python segue o vínculo da instância para a classe, se um atributo não existe em uma instância. Isso é tudo que há sobre *herança* no Python: ela ocorre no momento da qualificação do atributo e envolve apenas pesquisa de nomes em objetos vinculados (por exemplo, seguindo os vínculos *é um* na Figura 20-1).

Na função `setdata` dentro de `FirstClass`, o valor passado é atribuído a `self.data`. Dentro de um método, `self` – o nome dado ao argumento mais à esquerda, por convenção – refere-

se automaticamente à instância que está sendo processada (x ou y); portanto, as atribuições armazenam valores nos espaços de nome das instâncias e não na classe (é assim que os nomes data da Figura 20-1 são criados).

Como as classes geram várias instâncias, os métodos precisam passar pelo argumento `self` para chegar à instância a ser processada. Quando chamamos o método `display` da classe para imprimir `self.data`, vemos que ele é diferente em cada instância; por outro lado, `display` é o mesmo em x e y, pois é proveniente (é herdado) da classe:

```
>>> x.display()                    # self.data difere em cada um.
King Arthur
>>> y.display()
3.14159
```

Note que armazenamos tipos de objeto diferentes no membro `data` (uma string e um número de ponto flutuante). Assim como tudo no Python, não existem declarações para atributos de instância (às vezes chamados de membros); eles começam a existir na primeira vez que recebem um valor, exatamente como as variáveis simples. Na verdade, podemos alterar atributos de instância na própria instrução `class`, atribuindo a `self` em métodos, ou fora da classe, atribuindo a um objeto instância explícito:

```
>>> x.data = "New value"           # Também pode obter/configurar atributos
>>> x.display()                    # fora da classe.
New value
```

Embora seja menos comum, poderíamos até gerar um novo atributo na instância, atribuindo a seu nome fora das funções de método da classe:

```
>>> x.anothername = "spam"         # Pode obter/configurar atributos
```

Isso anexaria um novo atributo, chamado `anothername`, no objeto instância x, o qual poderia ou não ser usado por qualquer um dos métodos da classe. Normalmente, as classes criam todos os atributos da instância por meio de atribuição ao argumento `self`, mas não são obrigadas a fazer isso. Os programas podem buscar, alterar ou criar atributos em qualquer objeto para o qual tenham uma referência.

AS CLASSES SÃO PERSONALIZADAS POR MEIO DE HERANÇA

Além de servirem como geradoras de objetos, as classes também nos permitem fazer alterações por meio da introdução de novos componentes (chamados de subclasses), em vez de alterar no local os componentes já existentes. Os objetos instância gerados a partir de uma classe herdam os atributos da classe. O Python também permite que as classes herdem de outras classes e isso abre a porta para o desenvolvimento de *hierarquias* de classes, que especializam comportamento anulando atributos mais abaixo na hierarquia. Aqui, também, não há nenhum correspondente nos módulos: seus atributos residem em um único espaço de nome simples.

No Python, as instâncias herdam de classes e as classes herdam de superclasses. Aqui estão as principais idéias por trás dos mecanismos de herança de atributos:

As superclasses são listadas entre parênteses no cabeçalho de uma instrução class. Para herdar atributos de outra classe, basta listar a classe entre parênteses no cabeçalho de uma instrução `class`. A classe que herda é chamada de *subclasse* e a classe da qual os atributos são herdados é chamada de *superclasse*.

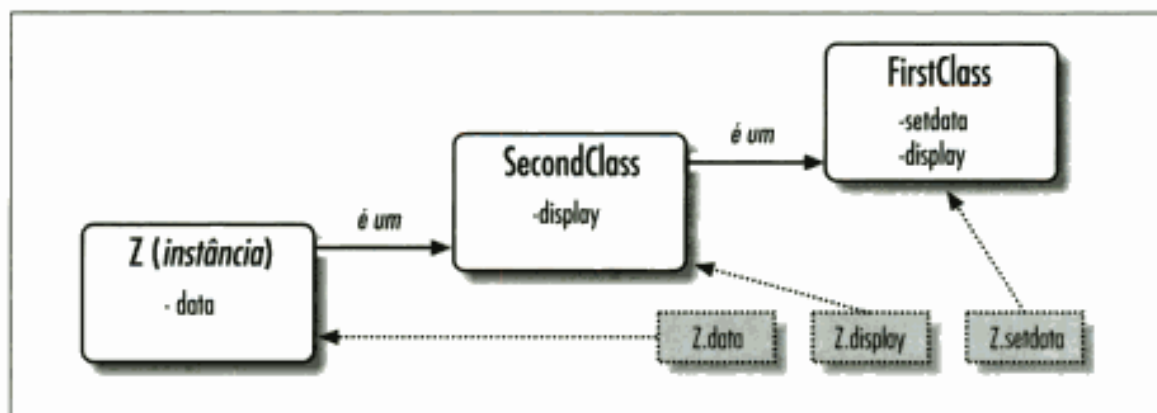



Figura 20-2 Especialização por meio da anulação de nomes herdados.

Como antes, produzimos um objeto instância de `SecondClass`, chamando-a. A chamada de `setdata` ainda executa a versão de `FirstClass`, mas desta vez o atributo `display` é proveniente de `SecondClass` e imprime uma mensagem personalizada.

Aqui está algo muito importante a notar sobre a POO: a especialização introduzida em `SecondClass` é completamente *externa* a `FirstClass`; ela não afeta objetos de `FirstClass` já existentes ou futuros – assim como `x` no exemplo anterior:

```
>>> x.display()    # x ainda é uma instância de FirstClass (mensagem antiga).
New value
```

Em vez de alterar `FirstClass`, a personalizamos. Naturalmente, este é um exemplo artificial, mas como regra, como a herança nos permite fazer alterações como essa em componentes externos (isto é, em subclasses), as classes freqüentemente suportam extensão e reutilização melhor do que as funções ou os módulos.

Classes e módulos

Antes de prosseguirmos, lembre-se de que não há nada de mágico a respeito de um nome de classe. Trata-se apenas de uma variável atribuída a um objeto quando a instrução `class` é executada e o objeto pode ser referenciado com qualquer expressão normal. Por exemplo, se nossa `FirstClass` fosse escrita em um arquivo de módulo, em vez de ser digitada interativamente, poderíamos importar e usar seu nome normalmente em uma linha de cabeçalho de classe:

```
from modulenome import FirstClass       # Copia o nome em meu escopo.
class SecondClass(FirstClass):          # Usa o nome da classe diretamente.
    def display(self): ...
```

Ou, equivalentemente:

```
import modulename                              # Acessa o módulo inteiro.
class SecondClass(modulename.FirstClass):      # Qualifica para referenciar
    def display(self): ...
```

Assim como tudo mais, os nomes de classe sempre residem dentro de um módulo e, assim, seguem todas as regras que estudamos na Parte V. Por exemplo, mais de uma instrução `class` pode ser escrita em um único arquivo de módulo – assim como outros nomes em um módulo, elas são executadas e definidas durante importações e tornam-se atributos de módulo distintos. Em geral, cada módulo pode misturar arbitrariamente qualquer número de variáveis, funções e classes, e todos os nomes em um módulo se comportam da mesma maneira. O arquivo *food.py* demonstra isso:

```
var = 1             # food.var
def func():         # food.func
    ...
class spam:         # food.spam
    ...
class ham:          # food.ham
    ...
class eggs:         # food.eggs
   ...
```

Isso vale mesmo que aconteça de o módulo e a classe terem o mesmo nome. Por exemplo, dado o seguinte arquivo, *person.py*:

```
class person:
    ...
```

Precisamos passar pelo módulo para buscar a classe, como sempre:

```
import person                 # Importa o módulo
x = person.person()           # classe dentro do módulo.
```

Embora esse caminho possa parecer redundante, ele é exigido: `person.person` refere-se à classe `person` dentro do módulo `person`. Escrever apenas `person` obtém o módulo e não a classe, a menos que a instrução `from` seja usada:

```
from person import person     # Obtém a classe a partir do módulo.
x = person()                  # Usa o nome da classe.
```

Assim como as outras variáveis, nunca podemos ver uma classe em um arquivo sem primeiro importar e, de alguma forma, buscar a partir de seu arquivo envolvente. Se isso parece confuso, não use o mesmo nome para um módulo e uma classe dentro dele.

Lembre-se também de que, embora as classes e os módulos sejam ambos espaços de nome para anexar atributos, eles correspondem a estruturas de código-fonte muito diferentes: um módulo reflete um arquivo inteiro, mas uma classe é uma instrução dentro de um arquivo. Falaremos mais sobre essas distinções posteriormente, nesta parte do livro.

AS CLASSES PODEM INTERCEPTAR OPERADORES DO PYTHON

Vamos ver a terceira distinção importante das classes: a sobrecarga de operadores. Em termos simples, a sobrecarga de operadores permite que objetos desenvolvidos com classes interceptem e respondam às operações que funcionam em tipos internos: adição, fracionamento, impressão, qualificação etc. Trata-se principalmente de um mecanismo de envio automático: as expressões direcionam o controle para implementações em classes. Aqui também não há nada semelhante nos módulos: os módulos podem implementar chamadas de função, mas não o comportamento de expressões.

Embora pudéssemos implementar todo comportamento de classe como funções de método, a sobrecarga de operadores permite que os objetos sejam mais fortemente integrados com o modelo de objeto do Python. Além disso, como a sobrecarga de operadores faz nossos próprios objetos atuarem como internos, ela tende a promover interfaces de objeto mais consistentes e mais fáceis de aprender. Aqui estão as principais idéias existentes por trás da sobrecarga de operadores:

Os métodos com nomes como `__X__` ***são ganchos especiais.*** A sobrecarga de operadores do Python é implementada por meio do fornecimento de métodos nomeados de forma especial para interceptar operações.

Tais métodos são chamados automaticamente quando o Python avalia operadores. Por exemplo, se um objeto herda um método `__add__`, o método é chamado quando o objeto aparece em uma expressão `+`.

As classes podem anular a maioria das operações de tipo interno. Existem dezenas de nomes de método de operador especiais, para interceptar e implementar praticamente toda operação disponível para tipos internos.

Os operadores permitem que as classes sejam integradas com o modelo de objeto do Python. Sobrecarregando operações de tipo, os objetos definidos pelo usuário implementados com classes atuam exatamente como os internos e, assim, fornecem consistência.

Um terceiro exemplo

Vamos ver outro exemplo. Desta vez, definimos uma subclasse de `SecondClass` que implementa três atributos de nome especial que o Python chamará automaticamente: `__init__` é chamado quando um novo objeto instância está sendo construído (`self` é o novo objeto `ThirdClass`), `__add__` e `__mul__` são chamados quando uma instância de `ThirdClass` aparece em expressões `+` e `*`, respectivamente:

```
>>> class ThirdClass(SecondClass):                   # é um SecondClass
...     def __init__(self, value):                   # Em "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):                    # Em "self + other"
...         return ThirdClass(self.data + other):
...     def __mul__(self, other):
...         self.data = self.data * other            # Em "self * other"

>>>     a = ThirdClass("abc")                        # Novo __init__ chamado
>>>     a.display()                                  # Método herdado
Current value = "abc"

>>> b = a + 'xyz'                       # Novo __add__: cria uma nova instância
>>> b.display()
Current value = "abcxyz"

>>> a * 3                               # Novo __mul__: altera instância no local
>>> a.display()
Current value = "abcabcabc"
```

`ThirdClass` é uma `SecondClass`; portanto, suas instâncias herdam display de `SecondClass`. Mas agora as chamadas de geração de `ThirdClass` passam um argumento (por exemplo, "abc"); ele é passado para o argumento value no construtor `__init__` e, lá, é atribuído a `self.data`. Além disso, os objetos de `ThirdClass` podem aparecer em expressões `+` e `*`; o Python passa o objeto instância à esquerda para o argumento `self` e o valor à direita para `other`, conforme ilustrado na Figura 20-3.

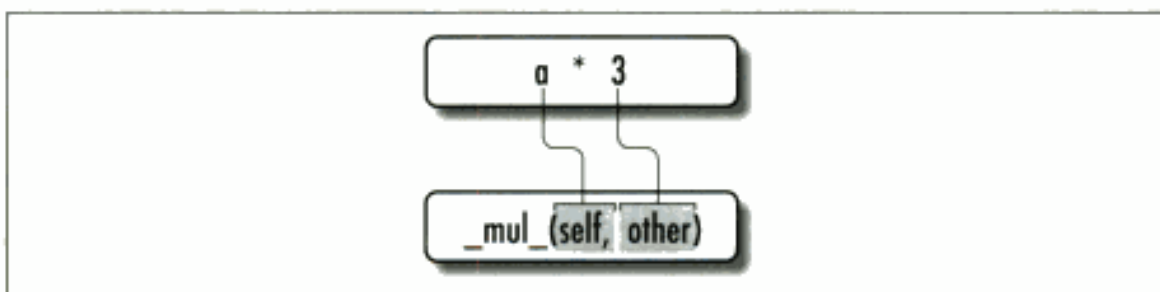



Figura 20-3 Os operadores são mapeados para métodos especiais.

Os métodos de nome especial, como `__init__` e `__add__`, são herdados pelas subclasses e pelas instâncias, exatamente como qualquer outro nome atribuído em uma instrução `class`. Se os métodos não estão escritos em uma classe, o Python procura esses nomes em todas as superclasses, como sempre. Os nomes de método de sobrecarga de operador também não são palavras internas nem reservadas: são apenas atributos que o Python procura quando os objetos aparecem em vários contextos. Normalmente, eles são chamados pelo Python automaticamente, mas ocasionalmente também podem ser chamados pelo seu código.

Note que o método `__add__` produz e retorna um *novo* objeto instância de sua classe (chamando `ThirdClass` com o valor do resultado), mas `__mul__` *altera* no local o objeto instância corrente (reatribuindo um atributo `self`). Isso é diferente do comportamento dos tipos internos, como números e strings, que sempre produzem um novo objeto para o operador `*`. Como a sobrecarga de operadores é, na verdade, apenas um mecanismo de envio de expressão para método, você pode interpretar os operadores como quiser em seus próprios objetos classe.*

Por que sobrecarga de operador?

Como desenvolvedor de classes, você pode optar por usar sobrecarga de operador ou não. Sua escolha depende simplesmente do quanto você deseja que seu objeto se pareça e se comporte como um tipo interno. Se você omitir um método de sobrecarga de operador e não herdá-lo de uma superclasse, a operação correspondente não será suportada para suas instâncias e simplesmente lançará uma exceção (ou usará um padrão previamente estabelecido), se for tentada.

Francamente, muitos métodos de sobrecarga de operador tendem a ser usados somente na implementação de objetos que são matemáticos por natureza; uma classe de vetor ou matriz pode sobrecarregar a adição, por exemplo, mas uma classe de funcionários provavelmente não poderia. Para classes mais simples, você poderia não usar sobrecarga e, em vez disso, contar com chamadas de método explícitas para implementar o comportamento de seu objeto.

Por outro lado, você também poderia usar sobrecarga de operador se precisasse passar um objeto definido pelo usuário para uma função desenvolvida de forma a esperar os operadores disponíveis em um tipo interno, como uma lista ou um dicionário. Implementando o mesmo conjunto de operadores em sua classe, seus objetos suportarão a mesma interface de objeto esperada e, assim, serão compatíveis com a função.

Normalmente, um certo método de sobrecarga aparece em quase toda classe real: o método construtor `__init__`. Como ele permite que as classes preencham imediatamente os atributos em suas instâncias recentemente criadas, o construtor é útil para quase todo tipo de classe que você possa desenvolver. Na verdade, mesmo que os atributos de instância não sejam declarados no Python, normalmente você pode descobrir quais atributos uma instância terá, inspecionando o código do método `__init__` de sua classe. Vamos ver mais técnicas de herança e sobrecarga de operador em ação, no Capítulo 21.

* Mas você provavelmente não deve fazer isso. A prática comum prescreve que operadores sobrecarregados devem funcionar da mesma maneira que as implementações do operador interno. Neste caso, isso quer dizer que nosso método `__mul__` deve retornar um *novo* objeto como resultado, em vez de alterar a instância (`self`) no local. Para alterações no local, uma chamada de método `mul` pode ser um estilo melhor do que uma sobrecarga de `*` aqui (por exemplo, `a.mul(3)`, em vez de `* 3`).

21

Detalhes do Desenvolvimento de Classes

Tudo que foi visto no Capítulo 20 fez sentido? Se não fez, não se preocupe. Agora que demos um rápido passeio, vamos nos aprofundar um pouco mais e estudar com mais detalhes os conceitos que apresentamos. Este capítulo dá um segundo passo para formalizar e expandir algumas das idéias do desenvolvimento de classes apresentadas no Capítulo 20.

A INSTRUÇÃO CLASS

Embora, superficialmente, a instrução `class` do Python pareça semelhante à de outras linguagens de POO, inspecionando-a mais de perto, vemos que ela é bastante diferente da que alguns programadores estão acostumados. Por exemplo, como na linguagem C++, a instrução `class` é a principal ferramenta de POO do Python. Ao contrário da linguagem C++, a instrução `class` do Python não é uma declaração. Assim como a instrução `def`, a instrução `class` é uma construtora de objetos e uma atribuição implícita – quando executada, ela gera um objeto classe e armazena uma referência para ele no nome usado no cabeçalho. Também como a instrução `def`, a instrução `class` é código realmente executável – sua classe não existe até que o Python atinja e execute a instrução `class` (normalmente, ao importar o módulo em que está escrito, mas até então, não).

Forma geral

A instrução `class` é uma instrução composta, normalmente contendo um miolo com instruções endentadas sob ela. No cabeçalho, são listadas as superclasses, entre parênteses e separadas por vírgulas, após o nome da classe. Listar mais de uma superclasse leva à herança múltipla (sobre o que falaremos mais, no próximo capítulo). Aqui está a forma geral da instrução:

```
class <nome>(superclasse,...):          # Atribui ao nome.
    data = valor                        # Dados de classe compartilhados
    def método(self...):                # Métodos
        self.membro = valor             # Dados por instância
```

Dentro da instrução `class`, toda atribuição gera um atributo de classe e operadores de sobrecarga de métodos com nomes especiais. Por exemplo, uma função denominada `__init__` é chamada no momento da construção do objeto instância, se for definida.

Exemplo

As classes são principalmente apenas espaços de nome – uma ferramenta para definir nomes (isto é, atributos) que exportam dados e lógica para clientes. Então, como você vai da instrução `class` para um espaço de nome?

Aqui está como. Assim como acontece com os arquivos de módulo, as instruções aninhadas no miolo de uma instrução `class` criam seus atributos. Quando o Python executa uma instrução `class` (não uma chamada para uma classe), ela executa todas as instruções de seu miolo, de cima para baixo. As atribuições que acontecem durante esse processo criam nomes no escopo local da classe, os quais se tornam atributos no objeto classe associado. Por isso, as classes são semelhantes aos módulos e às funções:

- Assim como as funções, as instruções `class` são um escopo local onde nomes são criados por atribuições aninhadas.
- Assim como os módulos, os nomes atribuídos em uma instrução `class` tornam-se atributos em um objeto classe.

A principal distinção das classes é que seus espaços de nome também são a base da herança no Python; atributos são buscados de outras classes, se não forem encontrados em um objeto classe ou em um objeto instância.

Como a instrução `class` é composta, qualquer tipo de instrução pode ser aninhada dentro de seu miolo – `print`, `=`, `if`, `def` etc. Todas as instruções dentro da instrução `class` são executadas quando ela é executada (e não quando a classe é chamada posteriormente para produzir uma instância). Todo nome atribuído dentro da instrução `class` produz um atributo de classe. As instruções `def` aninhadas produzem métodos de classe, mas outras atribuições também produzem atributos. Por exemplo:

```
>>> class SharedData:
...     spam = 42                   # Gera um atributo de classe.
...
>>> x = SharedData()                # Produz duas instâncias.
>>> y = SharedData()
>>> x.spam, y.spam                  # Elas herdam e compartilham spam.
(42, 42)
```

Aqui, como o nome `spam` é atribuído no nível superior de uma instrução `class`, ele é anexado à classe e, assim, será compartilhado por todas as instâncias. Altere-o passando pelo nome da classe; refira-se a ele por meio de instâncias ou da classe.*

```
>>> SharedData.spam = 99
>>> x.spam, y.spam, SharedData.spam
(99, 99, 99)
```

* Se você já usou a linguagem C++, talvez reconheça isso como semelhante à noção de dados de classe "estáticos" daquela linguagem – membros armazenados na classe, independentes das instâncias. No Python, não há nada de especial: todos os atributos de classe são apenas nomes atribuídos na instrução `class`, seja para referenciar funções ("métodos" da linguagem C++) ou qualquer outra coisa ("membros" do C++).

Tais atributos de classe podem ser usados para gerenciar informações que abrangem todas as instâncias – um contador do número de instâncias geradas, por exemplo (expandiremos essa idéia no Capítulo 23). Agora, veja o que acontece se atribuímos o nome `spam` por meio de uma instância, em vez da classe:

```
>>> x.spam = 88
>>> x.spam, y.spam, SharedData.spam
(88, 99, 99)
```

As atribuições a atributos de instância criam ou alteram esse nome na instância, em vez da classe compartilhada. Em geral, a pesquisa de herança ocorre apenas na *referência* do atributo e não na atribuição: a atribuição para o atributo de um objeto sempre altera esse objeto e não outro.* Por exemplo, `y.spam` é pesquisado na classe pela herança, mas a atribuição a `x.spam` anexa um nome no próprio `x`.

Quando produzimos instâncias dessa classe, o nome `data` também é anexado às instâncias pela atribuição a `self.data` no método construtor:

```
class MixeNames:                              # Define a classe.
    data = 'spam'                             # Atribui atributo da classe.
    def __init__(self, value):                # Atribui nome de método.
        self.data = value                     # Atribui atributo de instância.
    def display(self):
        print self.data, MixedNames.data      # Atributo de instância,
                                              atributo de classe
```

Essa classe contém duas instruções `def`, as quais vinculam atributos de classe a funções de método. Ela também contém uma instrução de atribuição `=`; como essa atribuição em nível de classe atribui o nome `data` dentro da instrução `class`, ele fica no escopo local da classe e torna-se um atributo do objeto classe. Assim como todos os atributos de classe, `data` é herdado e compartilhado por todas as instâncias da classe que não têm seu próprio atributo `data`.

Quando produzimos instâncias dessa classe, o nome `data` também é anexado às instâncias pela atribuição a `self.data` no método construtor:

```
>>> x = MixedNames(1)          # Produz dois objetos instância.
>>> y = MixedNames(2)          # Cada um tem seus próprios dados.
>>> x.display(); y.display()   # self.data difere, Subclass.data é o mesmo.
1 spam
2 spam
```

O resultado é que `data` reside em dois lugares: em objetos instância (criados pela atribuição `self.data` em `__init__`) e na classe da qual herdam nomes (criada pela atribuição de `data` na instrução `class`). O método `display` da instrução `class` imprime as duas versões, primeiro qualificando a instância de `self` e depois a classe.

Usando essas técnicas para armazenar atributos em diferentes objetos, você determina seu escopo e sua visibilidade. Quando anexados às classes, os nomes são compartilhados; nas instâncias, os nomes registram dados por instância e não comportamento ou dados compartilhados. Embora a herança pesquise nomes para nós, sempre podemos chegar a um atributo em qualquer parte de uma árvore, acessando diretamente o objeto desejado.

* A menos que a operação de atribuição de atributo tenha sido redefinida com o método de sobrecarga de operador `__setattr__` para fazer algo exclusivo.

No exemplo, `x.data` e `self.data` escolherão um nome de instância, o qual normalmente oculta o mesmo nome na classe. Mas `MixedNames.data` pega o nome da classe explicitamente. Veremos diversas funções para esses padrões de desenvolvimento posteriormente; a próxima seção descreve uma das mais comuns.

MÉTODOS

Como você já conhece as funções, também conhece os métodos nas classes. Os métodos são apenas objetos função criados por instruções `def` aninhadas no miolo de uma instrução `class`. De uma perspectiva abstrata, os métodos fornecem comportamento para os objetos instância herdarem. Da perspectiva da programação, os métodos funcionam exatamente como as funções simples, com uma exceção fundamental: o primeiro argumento sempre recebe o objeto instância que é o sujeito implícito de uma chamada de método.

Em outras palavras, o Python faz automaticamente o mapeamento das chamadas de método de instância nas funções de método de classe, como segue. As chamadas de método feitas por meio de uma instância:

```
instância.método(args...)
```

são automaticamente transformadas em chamadas de função de método desta forma:

```
classe.método (instância, args...)
```

onde a classe é determinada pela localização do nome do método usando o procedimento de pesquisa de herança do Python. Na verdade, as duas formas de chamada são válidas no Python.

Além da herança de nomes de atributo de método normal, o primeiro argumento especial é a única mágica real por trás das chamadas de método. Em um método de classe, normalmente o primeiro argumento é chamado de `self`, por convenção (tecnicamente, apenas sua posição é significativa e não seu nome). Esse argumento fornece métodos com um gancho de volta para a instância – como as classes geram muitos objetos instância, elas precisam usar esse argumento para gerenciar dados que variam de acordo com a instância.

Os programadores de C++ podem reconhecer o argumento `self` do Python como semelhante ao ponteiro `"this"` daquela linguagem. No Python, contudo, `self` está sempre explícito em seu código. Os métodos sempre devem passar por `self` para buscar ou alterar atributos da instância que está sendo processada pela chamada de método corrente. Essa natureza explícita de `self` é assim por design – a presença desse nome torna evidente que você está usando nomes de atributo em seu script e não um nome no escopo local ou global.

Exemplo

Vamos ver um exemplo. Suponha que definamos a seguinte classe:

```
class NextClass:                    # Define a classe.
    def printer(self, text):        # Define o método.
    self.message = text             # Altera a instância.
    print self.message              # Acessa a instância.
```

O nome `printer` referencia um objeto função; como ele é atribuído no escopo da instrução `class`, torna-se um atributo do objeto classe e é herdado por toda instância produzida a partir da classe. Normalmente, como métodos como `printer` são projetados para processar instâncias, os chamamos por meio de instâncias:

```
>>> x = NextClass()                 # Produz instância
```

O resultado é uma árvore de espaços de nome de atributo, a qual cresce a partir de uma instância, passando pela classe a partir da qual foi gerada, chegando até todas as superclasses listadas nos cabeçalhos de classe. O Python pesquisa para cima nessa árvore, das instâncias para as superclasses, sempre que você usa qualificação para buscar um nome de atributo de um objeto instância.*

Especializando métodos herdados

O modelo de herança com pesquisa de árvore, que acabamos de descrever, mostra-se uma maneira excelente de especializar sistemas. Como a herança encontra nomes em subclasses, antes de verificar as superclasses, as subclasses podem substituir comportamento padrão redefinindo os atributos da superclasse. Na verdade, você pode construir sistemas inteiros como hierarquias de classes, as quais são estendidas pela adição de novas subclasses externas, em vez de alterar no local a lógica já existente.

A idéia da redefinição de nomes herdados leva a uma variedade de técnicas de especialização. Por exemplo, as subclasses podem *substituir* completamente os atributos herdados, *fornecer* atributos que uma superclasse espera encontrar e *estender* métodos da superclasse chamando de volta para a superclasse, a partir de um método anulado. Já vimos a substituição em ação; aqui está um exemplo que mostra o funcionamento da extensão:

```
>>> class Super:
...     def method(self):
...     print 'in Super.method'
...
>>> class Sub(Super):
...     def method(self):                    # Anula method.
...         print 'starting Sub.method'      # Adicionar ações aqui.
...         Super.method(self):              # Executa a ação padrão.
...         print 'ending Sub.method'
...
```

As chamadas diretas de método de superclasse são o ponto crucial da questão aqui. A classe `Sub` substitui a função `method` de `Super` por sua própria versão especializada. Mas dentro da substituição, `Sub` chama de volta a versão exportada por `Super`, para executar o comportamento padrão. Em outras palavras, `Sub.method` apenas estende o comportamento de `Super.method`, em vez de substituí-lo completamente:

```
>>> x = Super()                # Cria uma instância de Super.
>>> x. method()                # Executa Super.method
in Super.method

>>> x = Sub()                  # Cria uma instância de Sub.
>>> x.method()                 # Executa Sub.method, que chama Super.method
starting Sub.method
in Super.method
ending Sub.method
```

Esse padrão de desenvolvimento de extensão também é comumente usado com construtores; consulte a seção "Métodos" para ver um exemplo.

* Essa descrição não está 100% completa, pois os atributos de instância e de classe também podem ser criados pela atribuição a objetos fora de instruções `class`. Mas isso é muito menos comum e, às vezes, mais propenso a erros (as alterações não são isoladas nas instruções `class`). No Python, todos os atributos são sempre acessíveis por padrão. Falaremos mais sobre a privacidade de nomes no Capítulo 23.

Técnicas de interface de classe

A extensão é apenas uma maneira de fazer interface com uma superclasse. O arquivo a seguir, *specialize.py*, define várias classes que ilustram diversas técnicas comuns:

`Super`
: Define uma função `method` e uma `delegate` que espera um membro `action` em uma subclasse

`Inheritor`
: Não fornece novos nomes; portanto, recebe tudo que está definido em `Super`

`Replacer`
: Anula a função `method` de `Super` com sua própria versão

`Extender`
: Personaliza a função `method` de `Super`, anulando e chamando de volta para executar o padrão

`Provider`
: Implementa o método `action` esperado pelo método `delegate` de `Super`

Estude cada uma dessas subclasses para ter uma idéia das diversas maneiras pelas quais elas personalizam sua superclasse comum:

```
class Super:
    def method(self):
        print 'in Super.method'          # Comportamento padrão
    def delegate(self):
        self.action()                    # Esperado para ser definido

class Inheritor(Super):                  # Método herdado literalmente.
    pass

class Replacer(Super):                   # Substitui o método completamente.
    def method(self):
        print 'in Replacer.method'

class Extender(Super):                   # Estende o comportamento de method.
    def method(self):
        print 'starting Extender.method'
        Super.method(self)
        print 'ending Extender.method'

class Provider(Super):                   # Preenche um método exigido.
    def action(self):
        print 'in Provider.action'

if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print '\n' + klass.__name__ + '...'
        klass().method()

    print '\nProvider...'
    x = Provider()
    x.delegate()
```

Vale a pena destacar algumas coisas aqui. O código de auto-teste no final desse exemplo cria instâncias de três classes diferentes em um loop `for`. Como as classes são objetos, você pode colocá-

las em uma tupla e criar instâncias genericamente (mais informações sobre essa idéia aparecerão posteriormente). As classes também têm o atributo especial `__name__`, como os módulos; ele é apenas previamente configurado com uma string contendo o nome no cabeçalho da classe:

```
% python specialize.py

Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

Superclasses abstratas

Observe o funcionamento da classe `Provider` no exemplo anterior. Quando chamamos o método `delegate`, por meio de uma instância de `Provider`, ocorrem *duas* pesquisas de herança independentes:

1. Na chamada inicial de `x.delegate`, o Python encontra o método `delegate` em `Super`, pesquisando na instância de `Provider` e acima. A instância `x` é passada para o argumento `self` do método, como sempre.
2. Dentro do método `Super.delegate`, `self.action` provoca uma nova pesquisa de herança independente, em `self` e acima. Como `self` referencia uma instância de `Provider`, o método `action` é localizado na subclasse `Provider`.

Essa espécie de estrutura de desenvolvimento tipo "preencher os espaços em branco" é típica dos modelos de POO. Pelo menos em termos do método `delegate`, nesse exemplo a superclasse é o que, às vezes, é chamada de *superclasse abstrata* – a classe que espera que partes de seu comportamento sejam fornecidas pelas subclasses. Se um método esperado não estiver definido em uma subclasse, o Python lançará uma exceção de nome indefinido, depois que a pesquisa de herança falhar. Às vezes, os desenvolvedores de classes tornam esses requisitos da subclasse mais evidentes com instruções `assert` ou lançando a exceção interna `NotImplementedError`:

```
class Super:
    def method(self):
        print 'in Super. method'
    def delegate(self):
        self.action()
    def action(self):
        assert 0, 'action must be defined!'
```

Vamos conhecer a instrução `assert` no Capítulo 24. Em resumo, se essa expressão for avaliada como falsa, lançará uma exceção com uma mensagem de erro. Aqui, a expressão é sempre falsa (0); portanto, vai gerar uma mensagem de erro se um método não for redefinido e a herança localizar a versão aqui. Como alternativa, algumas classes simplesmente lançam a exceção `NotImplemented` diretamente, em tais stubs de método. Vamos estudar a instrução `raise` no Capítulo 24.

Para ver um exemplo bem mais realista dos conceitos desta seção em ação, consulte o exercício "Hierarquia de animais no zoológico", no final da Parte VI e sua solução no Apêndice

Tabela 21-1 Métodos de sobrecarga de operador comuns (continuação)

Método	Sobrecarrega	Chamado por
`__repr__`,`__str__`	Impressão, conversões	`Print X, 'X', str(X)`
`__call__`	Chamadas de função	`X()`
`__getattr__`	Qualificação	`X.undefined`
`__setattr__`	Atribuição de atributo	`X.any = value`
`__getitem__`	Indexação	`X[key]`, loops `for`, testes `in`
`__setitem__`	Atribuição de índice	`X[key] = value`
`__len__`	Comprimento	`len(X)`, testes de verdade
`__cmp__`	Comparação	`X == Y, X < Y`
`__lt__`	Comparação específica	`X < Y` (ou então `__cmp__`)
`__eq__`	Comparação específica	`X == Y` (ou então `__cmp__`)
`__radd__`	Operador `'+'` no lado direito	Não-instância `+ X`
`__iadd__`	Adição no local (ampliada)	`X+=Y` (ou então `__add__`)
`__iter__`	Contextos de iteração	loops `for`, testes `in`, outros

Todos os métodos de sobrecarga têm nomes que começam e terminam com dois sublinhados, para distingui-los de outros nomes que você define em suas classes. O mapeamento do nome de método especial para expressão ou operação é simplesmente predefinido pela linguagem Python (e documentado no manual da linguagem padrão). Por exemplo, o nome `__add__` é sempre mapeado para expressões + pela definição da linguagem Python, independente do que o código do método `__add__` faz realmente.

Todos os métodos de sobrecarga de operador são opcionais – se você não escrever um, essa operação simplesmente não será suportada por sua classe (e poderá lançar uma exceção, se for tentada). A maioria dos métodos de sobrecarga só são usados em programas avançados que exigem que os objetos se comportem como internos. Entretanto, o construtor `__init__` tende a aparecer na maioria das classes. Já conhecemos o método construtor em tempo de inicialização `__init__` e alguns outros da Tabela 21-1. Como exemplo, vamos explorar alguns dos métodos adicionais da tabela.

__getitem__ intercepta referências de índice

O método `__getitem__` intercepta operações de indexação de instância. Quando uma instância `X` aparece em uma expressão de indexação, como `X[i]`, o Python chama um método `__getitem__` herdado pela instância (se houver), passando `X` para o primeiro argumento e o índice entre colchetes para o segundo argumento. Por exemplo, a classe a seguir retorna o quadrado de um valor de índice:

```
>>> class indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = indexer()
>>> X[2]                          # X[i] chama __getitem__(X, i).
4
>>> for i in range(5):
... print X[i],
...
0 1 4 9 16
```

__getimte__ e __iter__ implementam iteração

Aqui está um truque que nem sempre é óbvio para os iniciantes, mas que é incrivelmente útil: a instrução `for` funciona indexando repetidamente uma seqüência de zero até índices mais altos, até ser detectada uma exceção de fora do limite. Por isso, `__getitem__` também é uma maneira de sobrecarregar iteração no Python – se definidos, os loops `for` chamam o método `__getitem__` da classe em cada passagem, com deslocamento sucessivamente mais altos. Esse é um caso de "compre um, ganhe outro": qualquer objeto interno ou definido pelo usuário que responda à indexação também pode responder à iteração:

```
>>> class stepper:
... def __getitem__(self, i):
...     return self.data[i]
...
>>> X = stepper()                      # X é um objeto stepper.
>>> X.data = "Spam"
>>>
>>> X[1]                               # A indexação chama __getitem__.
'p'
>>> for item in X:                     # loops for chamam __getitem__.
...     print item,        # para itens de índices 0..N.
...
S p a m
```

De fato, trata-se na realidade de um caso de "compre um, ganhe muitos": qualquer classe que suporte loops `for` suporta automaticamente todos os contextos de iteração no Python, muitos dos quais já vimos em capítulos anteriores. Por exemplo, o teste de participação como membro `in`, abrangências de lista, a função interna `map`, atribuições de lista e tupla, e construtores de tipo, também chamarão `__getitem__` automaticamente, se definidos:

```
>>> 'p' in X                           # Tudo também chama __getitem__.
1

>>> [c for c in X]                     # Abrangência de lista
['S', 'p', 'a', 'm']

>>> map(None, X)                       # chamadas de map
['S', 'p', 'a', 'm']

>>> (a,b,c,d) = X                      # Atribuições de seqüência
>>> a, c, d
('S', 'a', 'm')

>>> list(X), tuple(X), ''.join(X)
(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')

>>> X
<__main__.stepper instance at 0x00A8D5D0>
```

Na prática, essa técnica pode ser usada para criar objetos que fornecem uma interface de seqüência e para adicionar lógica em operações de tipo de seqüência internos. Vamos rever essa idéia quando explorarmos os tipos internos no Capítulo 23.

Iteradores definidos pelo usuário

Atualmente, todos os contextos de iteração no Python tentarão primeiro encontrar um método `__iter__`, que é esperado para retornar um objeto que suporta o novo protocolo de iteração.

Se for fornecido, o Python chamará repetidamente o método `next` desse objeto para produzir itens, até que a exceção `StopIteration` seja lançada. Se esse método não for encontrado, o Python recorrerá ao esquema `__getitem__` e indexará repetidamente por deslocamentos, como antes, até uma exceção `IndexError` ser lançada.

No novo esquema, as classes implementam iteradores definidos pelo usuário, simplesmente implementando o protocolo iterador apresentado no Capítulo 14 para funções. Por exemplo, o arquivo a seguir, *iters.py*, estabelece um iterador definido pelo usuário que gera quadrados:

```
class Squares:
    def __init__(self, start, stop):
        self.value = start - 1
        self.stop = stop
    def __iter__(self):                 # Obtém objeto iterador
        return self
    def next(self):                     # em cada iteração de for.
        if self.value == self.stop:
            raise StopIteration
        self.value += 1
        return self.value ** 2
```

```
% python
>>> from iters import Squares
>>> for i in Squares(1,5):
...     print i,
...
1 4 9 16 25
```

Aqui, o objeto iterador é simplesmente a instância, `self`, pois o método `next` faz parte dessa classe. O fim da iteração é sinalizado com uma instrução `raise` do Python (mais informações sobre o lançamento de exceções aparecerão na próxima parte deste livro).

Um desenvolvimento equivalente com `__getitem__` poderia ser menos natural, pois a instrução `for` iteraria pelos deslocamentos zero e superiores; os deslocamentos passados seriam apenas indiretamente relacionados ao intervalo de valores produzido (`0..N` precisaria ser mapeado para `start..stop`). Como os objetos `__iter__` mantêm explicitamente o estado gerenciado entre as chamadas de `next`, eles podem ser mais gerais do que `__getitem__`.

Por outro lado, às vezes, os iteradores baseados em `__iter__` podem ser mais complexos e menos convenientes do que `__getitem__`. Eles são projetados para iteração e não para indexação aleatória. Na verdade, eles não sobrecarregam a expressão de indexação:

```
>>> X = Squares(1,5)
>>> X[1]
AttributeError: Squares instance has no attribute'__getitem__'
```

O esquema `__iter__` implementa os outros contextos de iteração que vimos em ação para `__getitem__` (testes de participação como membro, construtores de tipo, atribuição de seqüência etc.) Entretanto, ao contrário de `__getitem__`, `__iter__` é projetado para uma única varredura e não para muitas. Por exemplo, a classe `Squares` realiza apenas uma iteração; uma vez iterada, ela está vazia. Você precisa fazer um novo objeto iterador para cada nova iteração:

```
>>> X = Squares(1,5)
>>> [n for n in X]                      # Acaba com os itens
[1, 4, 9, 16, 25]
>>> [n for n in X]                      # Agora está vazia.
```

```
[]
>>> [n for n in Squares(1,5)]
[1, 4, 9, 16, 25]
>>> list(Squares(1,3))
[1, 4, 9]
```

Para ver mais detalhes sobre os iteradores, consulte o Capítulo 14. Note que esse exemplo provavelmente seria mais simples se fosse desenvolvido com *funções geradoras* – um assunto apresentado no Capítulo 14 e relacionado aos iteradores:

```
>>> from __future__ import generators # Necessário na versão 2.2
>>>
>>> def gsquares(start, stop):
...     for i in range(start, stop+1):
...         yield i ** 2
...
>>> for i in gsquare(1, 5):
...     print i,
...
1 4 9 16 25
```

Ao contrário da classe, a função salva automaticamente seu estado entre iterações. Contudo, as classes podem ser melhores na modelagem de iterações mais complexas, especialmente quando podem tirar proveito de hierarquias de herança. É claro que para esse exemplo artificial, você também poderia evitar as duas técnicas e usar simplesmente um loop `for`, `map` ou abrangência de lista para construir a lista toda de uma vez. A melhor e mais rápida maneira de executar uma tarefa no Python freqüentemente também é a mais simples:

```
>>> [x ** 2 for x in range(1, 6)]
[1, 4, 9, 16, 25]
```

__getattr__ e __setattr__ capturam referências de atributo

O método `__getattr__` intercepta qualificações de atributo. Mais especificamente, ele é chamado com o nome do atributo como uma string, quando você tenta qualificar uma instância em um nome de atributo *indefinido* (inexistente). Ele não será chamado se o Python puder encontrar o atributo usando seu procedimento de pesquisa de árvore de herança. Por causa de seu comportamento, `__getattr__` é útil como um gancho para responder aos pedidos de atributo de maneira genérica. Por exemplo:

```
>>> class empty:
...     def __getattr__(self, attrname):
...         if attrname == "age":
...             return 40
...         else:
...             raise AttributeError, attrname
...
>>> X = empty()
>>> X.age
40
>>> X.name
... texto do erro omitido...
AttributeError: name
```

Aqui, a classe `empty` e sua instância `X` não possuem atributos reais; portanto, o acesso a `X.age` é direcionado para o método `__getattr__`; `self` recebe a instância de (`X`) e `attrname` recebe

a string de nome de atributo indefinido ("age"). A classe faz com que age pareça um atributo real, retornando um valor real como resultado da expressão de qualificação X.age (40). Na verdade, age torna-se um *atributo calculado dinamicamente*.

Para outros atributos que a classe não sabe manipular, ela lança a exceção interna AttributeError para dizer ao Python que esse é um nome indefinido genuíno; solicitar X.name gera o erro. Você vai ver __getattr__ novamente, quando mostrarmos delegação e propriedades em funcionamento nos dois próximos capítulos, e falaremos mais sobre exceções na Parte VII.

Um método de sobrecarga relacionado, __setattr__, intercepta *todas* as atribuições de atributo. Se esse método estiver definido, self.attr=value se tornará self.__setattr__('attr',value). Isso é um pouco mais complicado de usar, pois atribuir a quaisquer atributos self dentro de __setattr__ chama __setattr__ novamente, causando um loop de recursividade infinito (e, finalmente, uma exceção de estouro de pilha!). Se você quiser usar esse método, certifique-se de que ele atribua quaisquer atributos de instância, indexando o dicionário de atributos, discutido na próxima seção. Use self.__dict__['name']=x e não self.name=x:

```
>>> class accesscontrol:
...     def __setattr__(self, attr, value):
...         if attr == 'age':
...             self.__dict__[attr] = value
...         else:
...             raise AttributeError, attr + ' not allowed'
...
>>> X = accesscontrol()
>>> X.age = 40                          # Chama __setattr__
>>> X.age
40
>>> X.name = 'mel'
... texto omitido...
AttributeError: name not allowed
```

Esses dois métodos de sobrecarga de acesso a atributo tendem a desempenhar funções altamente especializadas, algumas das quais conheceremos posteriormente neste livro. Em geral, elas permitem que você controle ou especialize o acesso aos atributos em seus objetos.

__repr__ e __str__ retornam representações de string

O próximo exemplo utiliza os métodos construtor __init__ e de sobrecarga __add__ que já vimos, mas também define um método __repr__ que retorna uma representação de string para instâncias. A formatação de string é usada para converter o objeto gerenciado self.data em uma string. Se definido, __repr__ (ou seu parente __str__) é chamado automaticamente, quando as instâncias de classe são impressas ou convertidas em strings; eles permitem que você defina para seus objetos uma string de impressão melhor do que a exibição padrão da instância.

```
>>> class adder:
...     def __init__(self, value=0):
...         self.data = value           # Inicializa dados.
...     def __data__(self, other):
...         self.data += other          # adiciona outros no local.
>>> class addrepr(adder):              # Herda __init__, __add__.
...     def __repr__(self):            # Adiciona representação de string.
...         return 'addrepr(%s)' % self.data    # Converte para string
                                                  como código.
```

```
>>> x = addrepr(2)                      # Executa __init__
>>> x +1                                # Executa __add__
>>> x                                   # Executa __repr__
addrepr(3)
>>> print x                             # Executa __repr__
addrepr(3)
>>> str(x), repr(x)                     # Executa __ repr__
(' addrepr(3)', ' addrepr(3)')
```

Então, por que dois métodos de exibição? A grosso modo, `__str__` é tentado primeiro para exibições amigáveis para o usuário, como a instrução `print` e a função interna `str`. O método `__repr__` deve, em princípio, retornar uma string que poderia ser usada como código executável para recriar o objeto e é usada para ecos do prompt interativo e para a função `repr`. O Python recorre a `__repr__` se não houver nenhum `__str__` presente, mas o inverso não é verdade:

```
>>> class addstr(adder):
...     def __str__(self):                  # __str__, mas não __repr__
...         return '[Value: %s]' % self.data # Converte para string amigável.
>>> x = addstr(3)
>>> x + 1
>>> x                                       # repr padrão
<__main__.addstr instance at 0x00B35EF0>
>>> print x                                 # Executa__str__
[Value: 4]
>>> str(x), repr(x)
('[Value: 4]', '<__main__.addstr instance at 0x00B35EF0>')
```

Por isso, `__repr__` pode ser melhor, se você quiser uma única exibição para todos os contextos. Contudo, definindo os dois métodos, você pode suportar diferentes exibições em diferentes contextos:

```
>>> class addboth(adder):
...     def __str__(self):
...         return '[Value: %s]' % self.data # String amigável para o usuário
...     def __repr__(self):
...         return 'addboth(%s)' % self.data # String como código

>>> x = addboth(4)
>>> x + 1
>>> x                                       # Executa __repr__
addboth(5)
>>> print x                                 # Executa __str__
[Value: 5]
>>> str(x), repr(x)
('[Value: 5]', 'addboth(5)')
```

__radd__ manipula adição no lado direito

Tecnicamente, o método `__add__` no exemplo anterior não suporta o uso de objetos instância no lado direito do operador +. Para implementar tais expressões e, assim, suportar operadores de estilo *comutativo*, escreva também o método `__radd__`. O Python chama `__radd__` somente quando o objeto no lado direito do operador + é sua instância de classe, mas o objeto no lado esquerdo não é uma instância de sua classe. Em vez disso, o método `__add__` do objeto no lado esquerdo é chamado em todos os outros casos:

```
...     def comp(self, other):
...         return self.value * other
...
>>> x = Prod(3)
>>> x.comp(3)
9
>>> x.comp(4)
12
```

Entretanto, `__call__` pode tornar-se mais útil ao fazer interface com APIs que esperam funções. Por exemplo, o kit de ferramentas de GUI Tkinter, que conheceremos posteriormente neste livro, permite registrar funções como rotinas de tratamento de eventos (também conhecidas como callbacks); quando ocorrem eventos, o Tkinter chama o objeto registrado. Se você quiser que uma rotina de tratamento de eventos mantenha o estado entre os eventos, pode registrar o método vinculado de uma classe ou uma instância compatível com a interface esperada, com `__call__`. Em nosso código, tanto `x.comp` do segundo exemplo quanto `x` do primeiro podem passar como objetos do tipo função, dessa maneira. Mais informações sobre métodos vinculados aparecerão no próximo capítulo.

__del__ é um destrutor

O construtor `__init__` é chamado quando uma instância é gerada. Seu complemento, o método destrutor `__del__`, é executado automaticamente quando o espaço de uma instância está sendo recuperado (isto é, no momento da "coleta de lixo"):

```
>>> class Life:
...     def __init__(self, name='unknown'):
...         print 'Hello', name
...         self.name = name
...     def __del__(self):
...         print 'Goodbye', self.name
...
>>> brian = Life('Brian')
Hello Brian
>>> brian = 'loretta'
Goodbye Brian
```

Aqui, quando `brian` recebe uma string, perdemos a última referência para a instância de `Life` e, assim, executamos seu método destrutor. Isso funciona e pode ser útil para implementar algumas atividades de limpeza, como terminar conexões de servidor. Entretanto, por diversos motivos os destrutores não são tão comumente usados no Python quanto em algumas linguagens de POO.

Por exemplo, como o Python recupera automaticamente todo espaço mantido por uma instância, quando a instância é recuperada, os destrutores não são necessários para gerenciamento de espaço.* Além disso, como você nem sempre pode prever facilmente quando uma instância será recuperada, freqüentemente é melhor desenvolver atividades de término em um método chamado explicitamente (ou em uma instrução `try/finally`, descrita na próxima parte

* Na implementação em C atual do Python, você também não precisa fechar objetos arquivos mantidos pela instância em destrutores, pois eles são fechados automaticamente, quando recuperados. Entretanto, conforme mencionado no Capítulo 7, é melhor chamar métodos de fechamento de arquivo explicitamente, pois o fechamento automático na recuperação é um recurso da implementação e não da linguagem em si (e pode variar sob o Jython).

As atribuições classificam nomes

Com procedimentos de pesquisa distintos para nomes qualificados e não qualificados, e várias camadas de pesquisa para ambos, às vezes pode ser confuso saber para onde um nome acabará indo. No Python, o lugar onde você atribui um nome é crucial – ele determina completamente em qual escopo e em qual objeto um nome residirá. O arquivo *manynames.py* ilustra e resume como isso se traduz em código:

```
X = 11                      # Nome/atributo de módulo (global)

class c:
    X = 22                  # Atributo de classe
    def m(self):
        X = 33              # Variável local no método
        self.X = 44         # Atributo de instância

def f():
    X = 55                  # Variável local em função

def g():
    print X                 # Acessa módulo X (11)
```

Como esse arquivo atribui o mesmo nome, `X`, em cinco locais diferentes, existem na verdade cinco `X` completamente diferentes nesse programa. De cima para baixo, as atribuições para nomes `X` geram um atributo de módulo, um atributo de classe, uma variável local em um método, um atributo de instância e um nome local em uma função.

Você deve estudar esse exemplo cuidadosamente, pois ele reúne idéias que exploramos nas últimas partes deste livro. Quando ele fizer sentido, você terá chegado ao nirvana do espaço de nome do Python. É claro que uma rota alternativa para o nirvana é simplesmente executar isso e ver o que acontece. Aqui está o restante desse arquivo, que produz uma instância e imprime todos os `X` que pode buscar:

```
obj = c()
obj.m()

print obj.X          # 44: instância
print c.X            # 22: classe (também conhecido como obj.X, se
                     não houver nenhum X na instância)
print X              # 11: módulo (também conhecido como manynames.X
                     # fora do arquivo)

#print c.m.X         # FALHA: visível apenas no método
#print f.X           # FALHA: visível apenas na função
```

Note que podemos passar pela classe para buscar seu atributo (`c.X`), mas nunca podemos buscar variáveis locais em funções nem em métodos fora de suas instruções `def`. As variáveis locais são visíveis para outro código somente dentro da instrução `def` e, na verdade, residem na memória apenas enquanto uma chamada para a função ou para o método está sendo executada.

Dicionários de espaços de nome

No Capítulo 16, aprendemos que os espaços de nome de módulo são implementados como dicionários e expostos com o atributo interno `__dict__`. O mesmo vale para objetos classe e instância: internamente, a qualificação de atributos é na verdade uma operação de indexação de dicionário, e a herança de atributos é apenas uma questão de pesquisar dicionários vinculados. De fato, dentro do Python, os objetos instância e classe são principalmente apenas

dicionários com vínculos. O Python expõe esses dicionários, assim como os vínculos entre eles, para uso em funções avançadas (por exemplo, para desenvolver ferramentas).

Para ajudá-lo a entender como os atributos funcionam internamente, vamos trabalhar em uma sessão interativa que acompanha como os dicionários de espaços de nome crescem quando classes estão envolvidas. Primeiro, vamos definir uma superclasse e uma subclasse com métodos que armazenarão dados em suas instâncias:

```
>>> class super:
...     def hello(self):
...         self.data1 = 'spam'
...
>>> class sub(super):
...     def hola(self):
...         self.data2 = 'eggs'
```

Quando criamos uma instância da subclasse, a instância começa com um dicionário de espaços de nome vazio, mas tem vínculos de volta para a classe para a pesquisa de herança seguir. Na verdade, a árvore de herança está explicitamente disponível em atributos especiais, os quais você pode inspecionar: as instâncias têm um atributo `__class__` que vincula com suas classes e as classes têm um atributo `__bases__` que é uma tupla contendo vínculos para superclasses mais altas:

```
>>> X = sub()
>>> X.__dict__
{}

>>> X.__class__
<class __main__.sub at 0x00A48448>

>>> sub.__bases__
(<class __main__.super at 0x00A3E1C8>,)

>>> super.__bases__
()
```

Quando as classes atribuem a atributos `self`, elas preenchem o objeto instância – isto é, os atributos acabam no dicionário de espaços de nome de atributo da instância e não nas classes. Os espaços de nome do objeto instância registram dados que podem variar de uma instância para outra e `self` é um gancho para esse espaço de nome:

```
>>> Y = sub()

>>> X.hello()
>>> X.__dict__
{'data1': 'spam'}

>>> X.hola()
>>> X.__dict__
{'data1': 'spam', 'data2', 'eggs'}

>>> sub.__dict__
{'__module__': '__main__', '__doc__': None, 'hola': <function hola at
0x00A47048>}

>>> super.__dict__
{'__module__': '__main__', 'hello: <function hello at 0x00A3C5A8>,
'__doc__': None}
```

```
>>> sub.__dict__.keys(), super.__dict__.keys()
(['__module__', '__doc__', 'hola'], ['__module__', 'hello', '__doc__'])

>>> Y.__dict__
{}
```

Observe os nomes com sublinhado extras nos dicionários de classe; eles são configurados automaticamente pelo Python. A maioria não é usada em programas típicos, mas algumas são utilizadas por ferramentas (por exemplo, `__doc__` contém as docstrings, discutidas no Capítulo 11).

Observe também que `Y`, uma segunda instância produzida no início dessa série, ainda tem um dicionário de espaços de nome vazio no final, mesmo que `X` tenha sido preenchido pelas atribuições nos métodos. Cada instância é um dicionário de espaços de nome independente, que começa vazio e pode registrar atributos completamente diferentes das outras instâncias da mesma classe.

Agora, como os atributos são na verdade chaves de dicionário dentro do Python, existem realmente duas maneiras de buscar e atribuir seus valores – por qualificação ou por indexação de chave:

```
>>> X.data1, X.__dict__['data1']
('spam', 'spam')

>>> X.data3 = 'toast'
>>> X.__dict__
{'data1': 'spam', 'data3': 'toast', 'data2': 'eggs'}

>>> X.__dict__['data3'] = 'ham'
>>> X.data3
'ham'
```

Contudo, essa equivalência só se aplica aos atributos vinculados à instância. Como a qualificação de atributo também realiza a *herança*, ela pode acessar atributos que a indexação do dicionário de espaços de nome não pode. O atributo herdado `X.hello`, por exemplo, não pode ser obtido por `X.__dict__['hello']`.

Finalmente, aqui está a função interna `dir`, que conhecemos nos capítulos 3 e 11, em funcionamento com objetos classe e instância. Essa função opera em tudo que tem atributos: `dir(objeto)` é semelhante a uma chamada `objeto.__dict__.keys()`. Observe, contudo, que `dir` ordena sua lista e inclui alguns atributos de sistema. A partir do Python 2.2, `dir` também coleta atributos *herdados* automaticamente.*

```
>>> X.__dict__
{'data1': 'spam', 'data3': 'ham', 'data2': 'eggs'}
>>> X.__dict__.keys()
['data1', 'data3', 'data2']

>>> dir(X)
['__doc__', '__module__', 'data1', 'data2', 'data3', 'hello', 'hola']
>>> dir(sub)
['__doc__', '__module__', 'hello', 'hola']
>>> dir(super)
['__doc__', '__module__', 'hello']
```

* O conteúdo dos resultados de dicionários de atributo e da chamada de `dir` podem mudar com o passar do tempo. Por exemplo, como agora o Python permite que tipos internos sejam colocados em subclasse como as classes, o conteúdo dos resultados de `dir` para tipos internos foi expandido para incluir métodos de sobrecarga de operador. Em geral, os nomes de atributo com sublinhados duplos no início e no fim são específicos do interpretador. Mais informações sobre subclasses de tipo aparecerão no Capítulo 23.

Aqui, a endentação marcada por pontos é usada para denotar a altura da árvore de classes. Podemos importar essas funções para onde quisermos uma exibição rápida de árvore de classes:

```
>>> class Emp: pass
>>> class Person(Emp): pass
>>> bob = Person()
>>> import classtree
>>> classtree.instancetree(bob)
Tree of <__main__.Person instance at 0x00AD34E8>
... Person
...... Emp
```

É claro que poderíamos melhorar esse formato de saída e talvez até esboçá-lo em uma tela de GUI. Independente de você desenvolver (ou usar) ou não essas ferramentas, este exemplo demonstra uma das muitas maneiras pelas quais podemos utilizar atributos especiais que expõem o funcionamento interno do interpretador. Conheceremos outra quando desenvolvermos uma classe de listagem de atributos de propósito geral, na seção sobre herança múltipla do Capítulo 22.

22

Projetando com Classes

Até aqui, nos concentramos na ferramenta de POO do Python – a classe. Mas a POO também está relacionada com os problemas de projeto – como usar classes para modelar objetos úteis. Esta seção tocará em algumas idéias básicas da POO e examinará alguns exemplos adicionais mais realistas do que os que foram mostrados até agora. Muitos dos termos de projeto mencionados aqui exigem mais explicação do que podemos dar; se esta seção aguçar sua curiosidade, sugerimos o estudo de um texto sobre projeto de POO ou sobre padrões de projeto como um próximo passo.

PYTHON E POO

A implementação de POO do Python pode ser resumida por três idéias:

Herança
: É baseada em pesquisa de atributos no Python (em expressões `X.nome`).

Polimorfismo
: Em `X.método`, o significado de método depende do tipo (classe) de `X`.

Encapsulamento
: Os métodos e os operadores implementam comportamento; a ocultação de dados é uma convenção, por padrão.

Agora, você já deve ter uma boa idéia do que é herança no Python. Já falamos sobre polimorfismo do Python algumas vezes; ele é proveniente da falta de declarações de tipo da linguagem. Como os atributos são sempre solucionados em tempo de execução, os objetos que implementam as mesmas interfaces são intercambiáveis. Os clientes não precisam saber que tipo de objeto está implementando um método que chamam.

Encapsulamento significa empacotamento no Python – ocultar os detalhes da implementação atrás da interface de um objeto. Conforme você vai ver no Capítulo 23, isso não significa privacidade forçada. O encapsulamento possibilita que a implementação da interface de um objeto seja alterada, sem afetar os usuários desse objeto.

Sobrecarga por assinaturas de chamada (ou não)

Algumas linguagens de POO também definem o polimorfismo com o significado de sobrecarga de funções com base nas assinaturas de tipo de seus argumentos. Como não há nenhuma declaração de tipo no Python, o conceito não se aplica; no Python, o polimorfismo é basedo em *interfaces* de objeto, e não em tipos. Por exemplo, você pode tentar sobrecarregar métodos por meio de suas listas de argumentos:

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

Esse código será executado, mas como a instrução `def` apenas atribui um objeto a um nome no escopo da classe, a última definição de uma função de método é a única mantida (é exatamente como se você escrevesse `X=1` e depois `X=2`; `X` será `2`).

As seleções baseadas em tipo sempre podem ser desenvolvidas usando-se as idéias de teste de tipo, que conhecemos no Capítulo 7, ou as ferramentas de lista de argumentos do Capítulo 13:

```
class C:
    def meth(self, *args):
        if len(args) == 1:
            ...
        elif type(arg[0]) == int:
            ...
```

Normalmente, contudo, você não deve fazer isso – conforme descrito no Capítulo 12, escreva seu código para esperar uma interface de objeto e não um tipo de dados específico. Desse modo, ele torna-se útil para uma categoria mais ampla de tipos e aplicações, agora e no futuro:

```
class C:
    def meth(self, x):
        x.operation()              # Presume que x faz a coisa certa.
```

Geralmente, também é considerado melhor usar nomes de método distintos para operações distintas, em vez de contar com assinaturas de chamada (independentemente da linguagem em que você desenvolver).

CLASSES COMO REGISTROS

O Capítulo 6 mostrou como você utiliza dicionários para registrar propriedades de entidades em seu programa. Vamos explorar isso com mais detalhes. Aqui está o exemplo de registros baseados em dicionário usado anteriormente:

```
>>> rec = {}
>>> rec['name'] = 'mel'
>>> rec['age'] = 40
>>> rec['job'] = 'trainer/writer'
>>>
>>> print rec[name']
mel
```

Esse código simula coisas como "registros" e "estruturas" em outras linguagens. Existem várias maneiras de fazer a mesma coisa com classes. Talvez a mais simples seja esta:

```
>>> class rec: pass
...
>>> rec.name = 'mel'
>>> rec.age = 40
>>> rec.job = 'trainer/writer'
>>>
>>> print rec.age
40
```

Esse código tem significativamente menos sintaxe do que o equivalente com dicionário. Ele usa uma instrução `class` vazia para gerar um objeto espaço de nome vazio (observe a instrução `pass` – precisamos de uma instrução sintaticamente igual, pois não há nenhuma lógica para desenvolver nesse caso). Uma vez feita a classe vazia, a preenchemos atribuindo a atributos de classe com o passar do tempo.

Isso funciona, mas precisaremos de uma nova instrução `class` para cada registro distinto que for necessário. Talvez mais normalmente, podemos, em vez disso, gerar *instâncias* de uma classe vazia para representar cada entidade distinta:

```
>>> class rec: pass
...
>>> pers1 = rec()
>>> pers1.name = 'mel'
>>> pers1.job = 'trainer'
>>> pers1.age = 40
>>>
>>> pers2 = rec()
>>> pers2.name = 'dave'
>>> pers2.job = 'developer'
>>>
>>> pers1.name, pers2.name
('mel', 'dave')
```

Aqui, fazemos dois registros a partir da mesma classe; as instâncias começam vazias, exatamente como as classes. Preenchemos o registro fazendo atribuições a atributos. Contudo, desta vez são dois objetos separados e, portanto, dois atributos `name` separados. De fato, instâncias da mesma classe nem mesmo precisam ter o mesmo conjunto de nomes de atributo; neste exemplo, um deles tem o nome exclusivo `age`. As instâncias são espaços de nome distintos – cada uma tem um dicionário de atributos distinto. Embora normalmente sejam preenchidas consistentemente por métodos de classe, elas são mais flexíveis do que você poderia esperar.

Finalmente, você poderia, em vez disso, desenvolver uma classe mais completa para implementar seu registro:

```
>>> class Person:
...     def __init__(self, name, job):
...         self.name = name
...         self.job = job
...     def info(self):
...         return (self.name, self.job)
...
>>> mark = Person('ml', 'trainer')
>>> dave = person('da', 'developer')
>>>
>>> mark.job, dave.info()
('trainer', ('da', 'developer'))
```

Esse esquema também produz múltiplas instâncias, mas desta vez a classe não é vazia: adicionamos lógica (métodos) para inicializar instâncias no momento da construção e coletar atributos em uma tupla. O construtor impõe alguma consistência nas instâncias aqui, sempre configurando os atributos `name` e `job`.

Eventualmente, poderíamos adicionar mais lógica para calcular salários, analisar nomes etc. Em última análise, poderíamos vincular a classe a uma hierarquia maior para herdar um conjunto de métodos, por meio de pesquisa de atributos automática de classes, e até armazenar instâncias da classe em um arquivo com conservação de objeto do Python para torná-las persistentes (mais informações sobre conservação e persistência aparecem em um quadro e também, posteriormente, no livro). No final, embora coisas como dicionários sejam flexíveis, as classes nos permitem adicionar comportamento em objetos de maneiras que os tipos internos e as funções simples não suportam diretamente.

POO E HERANÇA: RELACIONAMENTOS "É UM"

Já falamos com profundidade sobre os mecanismos de herança, mas gostaríamos de mostrar um exemplo de como ela pode ser usada para modelar relacionamentos do mundo real. Do ponto de vista do programador, a herança é iniciada pelas qualificações de atributo e provoca a pesquisa de um nome em uma instância, em sua classe e, depois, em suas superclasses. Do ponto de vista do projetista, a herança é uma maneira de especificar a participação como membro de um conjunto: uma classe define um conjunto de propriedades que podem ser herdados e personalizados por conjuntos mais específicos (isto é, subclasses).

Para ilustrar, vamos fazer funcionar aquele robô pizzaiolo sobre o qual falamos no início desta parte do livro. Suponha que tenhamos decidido explorar uma carreira alternativa e abrir uma pizzaria. Uma das primeiras coisas que precisaremos fazer é contratar empregados para atender os clientes, fazer a pizza etc. Sendo engenheiros por profissão, também decidimos construir um robô para fazer as pizzas, mas sendo política e ciberneticamente corretos, decidimos ainda tornar nosso robô um empregado plenamente habilitado, ganhando até um salário.

Nossa equipe da pizzaria pode ser definida pelas classes a seguir, no arquivo de exemplo *employees.py*. Ele define quatro classes e algum código de auto-teste. A classe mais geral, `Employee` (Empregado), fornece comportamento comum, como aumento de salários (`giveRaise`) e impressão (`__repr__`). Existem dois tipos de empregados e, assim, duas subclasses de `Employee`–`Chef` (Cozinheiro) e `Server` (Atendente). Ambas anulam o método herdado `work` para imprimir mensagens mais específicas. Finalmente, nosso robô pizzaiolo é modelado por uma classe ainda mais especializada: `PizzaRobot` é um tipo de `Chef`, que é um tipo de `Employee`. Em termos de POO, chamamos esses relacionamentos de vínculos "é um": um robô é um cozinheiro, que é um empregado.

```
class Employee:
    def __init__(self, name, salary=0)
        self.name = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print self.name, "does stuff"
    def __repr__(self):
        return "<Employee: name=%s, salary=%s>" % (self.name, self.salary)

class Chef(Employee):
```

Aqui, a classe `Uppercase` herda a lógica do loop de processamento de fluxo (e tudo mais que possa estar desenvolvido em suas superclasses). Ela precisa definir apenas o que é exclusivo a seu respeito – a lógica de conversão de dados. Quando esse arquivo é executado, ele produz e executa uma instância, a qual lê o arquivo *spam.txt* e grava o equivalente em letras maiúsculas desse arquivo no fluxo `stdout`:

```
C:\lp2e> type spam.txt
spam
spam
SPAM!
C:\lp2e> python converters.py
SPAM
SPAM
SPAM!
```

Para processar diferentes tipos de fluxos, passe diferentes tipos de objetos para a chamada de construção de classe. Aqui, usamos um arquivo de saída, em vez de um fluxo:

```
C:\lp2e> python
>>> import converters
>>> prog = converters.Uppercase(open('spam.txt'), open('spamup.txt', 'w'))
>>> prog.process()

C:\lp2e> type spamup.txt
SPAM
SPAM
SPAM!
```

Mas, conforme sugerido anteriormente, também poderíamos passar objetos arbitrários encerrados em classes que definissem as interfaces de método de entrada e saída exigidas. Aqui está um exemplo simples que passa uma classe gravadora que encerra o texto dentro de tags HTML:

```
C:\lp2e> python
>>> from converters import Uppercase
>>>
>>> class HTMLize:
...     def write(self, line):
...         print '<PRE>%s</PRE>' % line[:-1]
...
>>> Uppercase(open('spam.txt'), HTMLize()).process()
<PRE>SPAM</PRE>
<PRE>SPAM</PRE>
<PRE>SPAM!</PRE>
```

Se você acompanhar o fluxo de controle desse exemplo, verá que obtemos tanto a conversão para letras maiúsculas (por *herança*) como a formatação em HTML (por *composição*), mesmo que a lógica de processamento básica na superclasse original `Processor` não saiba nada sobre nenhuma das etapas. O código de processamento só tem o cuidado de ter um método `write` e de que um método chamado `convert` seja definido; ele não se preocupa com o que essas chamadas fazem. Tal polimorfismo e encapsulamento de lógica está por trás de grande parte do poder das classes.

No estado em que se encontra, a superclasse `Processor` fornece apenas um loop de varredura de arquivo. Em um trabalho mais real, poderíamos estendê-la para suportar mais ferramentas de programação para suas subclasses e, no processo, transformá-la em um *modelo* completo. Desenvolvendo-se tais ferramentas uma vez em uma superclasse, elas podem ser reutilizadas

em todos os seus programas. Mesmo neste exemplo simples, como muita coisa é empacotada e herdada com classes, tudo que precisaríamos desenvolver aqui seria a etapa da formatação em HTML. O resto é grátis.*

Por que isto é relevante: classes e persistência

Mencionamos a conservação algumas vezes nesta parte do livro, pois ela funciona particularmente bem com instâncias de classe. Por exemplo, além de nos permitirem simular interações do mundo real, as classes da pizzaria também poderiam ser usadas como base de um banco de dados de restaurantes persistente. As instâncias das classes podem ser armazenadas no disco em uma única etapa, usando-se os módulos `pickle` ou `shelve` do Python. A interface de conservação de objeto é notadamente fácil de usar:

```
import pickle
object = someClass()
file = open(filename, 'wb')       # Cria arquivo externo.
pickle.dump(object, file)         # Salva objeto no arquivo.

import pickle
file = open(filename, 'rb')
object = pickle.load(file)        # Busca-o de volta posteriormente.
```

A conservação converte os objetos que estão na memória em fluxos de bytes dispostos em série, os quais podem ser armazenados em arquivos, enviados por meio de uma rede etc. O inverso da conservação converte fluxos de bytes de volta para objetos idênticos na memória. O módulo `shelve` é semelhante, mas conserva os objetos automaticamente em um banco de dados de acesso pela chave, o qual exporta uma interface do tipo dicionário:

```
import shelve
object = someClass()
dbase = shelve.open('filename')
dbase['key'] = object             # Salva sob a chave

import shelve
dbase = shelve.open('filename')
object = dbase['key']             # Busca-o de volta posteriormente.
```

Em nosso exemplo, o uso de classes para modelar empregados significa que podemos obter um banco de dados de empregados e pizzarias simples, com pouco trabalho extra: a conservação de tais objetos instância em um arquivo os torna persistentes entre as execuções de programa no Python. Consulte o manual da biblioteca padrão e exemplos posteriores para obter mais informações sobre conservação.

POO E DELEGAÇÃO

Os programadores orientados a objetos freqüentemente falam sobre algo chamado delegação, que normalmente implica em objetos controladores que incorporam outros objetos, para os quais passam pedidos de operação. Os controladores podem cuidar de atividades administrativas, como o monitoramento de acessos etc. No Python, a delegação é freqüentemente implementada com o gancho de método `__getattr__`; como ele intercepta acessos a atributos inexistentes, uma classe envoltória pode usar `__getattr__` para direcionar acessos arbitrários para um objeto envolvido. Considere o arquivo *trace.py*, por exemplo:

* Para ver outro exemplo de composição em funcionamento, consulte o exercício desta parte chamado "Esboço do Papagaio Morto" e sua solução; ele é semelhante ao exemplo da pizzaria.

```
class wrapper:
    def __init__(self, object):
        self.wrapped = object                       # Salva o objeto.
    def __getattr__(self, attrname):
        print 'Trace:', attrname                    # Busca de Trace.
        return getattr(self.wrapped, attrname)      # Busca do delegado.
```

Lembre-se de que `__getattr__` obtém o nome de atributo como string. Esse código faz uso da função interna `getattr` para buscar um atributo do objeto envolvido pela string do nome – `getattr(X,N)` é como `X.N`, exceto que `N` é uma expressão avaliada como uma string no momento da execução e não uma variável. De fato, `getattr(X,N)` é semelhante a `X.__dict__[N]`, mas a primeira também realiza pesquisa de herança como `X.N` (veja a seção anterior sobre dicionários de espaços de nome).

Você pode usar a estratégia da classe `wrapper` desse módulo para gerenciar o acesso a qualquer objeto com atributos – listas, dicionários e até classes e instâncias. Aqui, a classe simplesmente imprime uma mensagem de rastreamento em cada acesso a atributo e delega o pedido de atributo para o objeto incorporado `wrapper`:

```
>>> from trace import wrapper
>>> x = wrapper([1,2,3])              # Encerra uma lista.
>>> x.append(4)                       # Delega para o método da lista.
Trace: append
>>> x.wrapped                         # Imprime meu membro.
[1, 2, 3, 4]

>>> x = wrapper({"a":1, "b": 2})      # Encerra um dicionário.
>>> x.keys()                          # Delega para o método do dicionário.
Trace: keys
['a', 'b']
```

Vamos rever as noções de objeto encerrado e operações delegadas como uma maneira de estender tipos internos, no Capítulo 23.

HERANÇA MÚLTIPLA

Na instrução `class`, mais de uma superclasse pode ser listada entre parênteses na linha de cabeçalho. Quando faz isso, você utiliza algo chamado *herança múltipla* – a classe e suas instâncias herdam nomes de todas as superclasses listadas.

Ao procurar um atributo, o Python pesquisa as superclasses do cabeçalho de classe, da esquerda para a direita, até encontrar uma correspondência. Tecnicamente, a pesquisa procede primeiro do fundo até o topo e depois da esquerda para a direita, pois qualquer uma das superclasses pode ter suas próprias superclasses.

Em geral, a herança múltipla é boa para modelar objetos pertencentes a mais de um conjunto. Por exemplo, alguém pode ser engenheiro, escritor, músico etc. e herdar propriedades de todos esses conjuntos.

Talvez a maneira mais comum de usar herança múltipla seja para "misturar" métodos de propósito geral de superclasses. Normalmente, tais superclasses são chamadas de *classes de mistura* – elas fornecem métodos que você adiciona em classes de aplicativo por herança. Por exemplo, a maneira padrão do Python imprimir um objeto instância de classe é incrivelmente útil:

```
>>> class Spam:
...     def __init__(self):           # Nenhum __repr__
```

```
...     def __init__(self):
...         self.data1 = 'food'
...
>>> X = Spam()
>>> X
<Instance of Spam, address 8821568:
        name data1=food
>
```

Agora, a classe `Lister` é útil para qualquer classe que você escreva – mesmo classes que já tenham uma superclasse. É aí que a herança múltipla mostra sua utilidade; adicionando (misturando) `Lister` na lista de superclasses em um cabeçalho de classe, você recebe seu método `__repr__` gratuitamente, enquanto ainda herda das superclasses existentes. O arquivo *testmixin.py* demonstra isso:

```
from mytools import Lister          # Obtém classe de ferramenta

class Super:
    def __init__(self):             # __init__ da superclasse
        self.data1 = "spam"

class Sub(Super, Lister):           # Mistura um __repr__
    def __init__(self):             # Lister tem acesso a self
        Super.__init__(self)
        self.data2 = "eggs"         # Mais atributos de instância
        self.data3 = 42

if __name__ == "__main__":
    X = Sub()
    print X                         # repr misturado
```

Aqui, `Sub` herda nomes de `Super` e de `Lister`; trata-se de uma composição de seus próprios nomes e dos nomes de suas duas superclasses. Quando você cria uma instância de `Sub` e a imprime, recebe automaticamente a representação mista personalizada de `Lister`:

```
C:lp2e> python testmixin.py
<Instance of Sub, address 7833392:
        name data3=42
        name data2=eggs
        name data1=spam
>
```

`Lister` funciona em qualquer classe em que seja misturada, pois `self` refere-se a uma instância da subclasse que serve-se de `Lister`, qualquer que seja ela. Se, posteriormente, você decidir estender o método `__repr__` de `Lister` para imprimir também todos os atributos de classe que uma instância herda, estará protegido; como se trata de um método herdado, alterar `Lister.__repr__` atualiza automaticamente a exibição de cada subclasse que importa a classe e a mistura.*

* Se você estiver curioso para saber como isso acontece, volte para a seção do Capítulo 21 sobre dicionários de espaços de nome, para ver dicas. Lá, vimos que as classes têm um atributo interno chamado `__bases__`, que é uma tupla dos objetos de superclasse da classe. Uma hierarquia de classes de propósito geral pode ir do atributo `__class__` de uma instância até sua classe e, depois, do atributo `__bases__` da classe até todas as superclasses recursivamente, de forma muito parecida com o exemplo `classtree.py` mostrado anteriormente. No Python 2.2 e posteriores, isso pode ser ainda mais simples: agora, a função interna `dir` inclui os nomes de atributo herdados automaticamente. Se você não se preocupa com a exibição da estrutura de árvore, pode apenas percorrer a lista de `dir`, em vez da lista de chaves de dicionário, e usar `getattr` para buscar atributos pela string de nome, em lugar da indexação de chaves de dicionário. Vamos apresentar essa idéia de uma nova forma em um exercício e em sua solução.

De certo modo, as classes de mistura são equivalentes aos módulos – pacotes de métodos úteis em uma variedade de clientes. Aqui está a classe `Lister` em funcionamento outra vez no modo de herança simples, nas instâncias de uma classe diferente; a POO promove a reutilização de código:

```
>>> from mytools import Lister
>>> class x(Lister):
...     pass
...
>>> t = x()
>>> t.a = 1; t.b = 2; t.c = 3
>>> t
<Instance of x, address 7797696:
        name b=2
        name a=1
        name c=3
>
```

As classes de mistura representam uma técnica poderosa. Na prática, a herança múltipla é uma ferramenta avançada e pode tornar-se complicada, se for usada sem cautela ou excessivamente. Assim como quase tudo mais na programação, ela pode ser um dispositivo útil, quando bem aplicada. Vamos rever esse assunto como um *problema* no final desta parte do livro. No Capítulo 23, conheceremos também uma opção (novas classes de estilo) que modifica a ordem de pesquisa para um caso de herança múltipla especial.

AS CLASSES SÃO OBJETOS: FÁBRICAS DE OBJETOS GENÉRICAS

Como as classes são objetos, é fácil fazê-las circular em um programa, armazená-las em estruturas de dados etc. Você também pode passar classes para funções que geram tipos de objetos arbitrários; às vezes, tais funções são chamadas de *fábricas* nos círculos de projeto em POO. Elas são um empreendimento maior em uma linguagem fortemente tipada, como a C++, mas são quase triviais no Python: a função `apply` e a sintaxe que conhecemos no Capítulo 14 podem chamar de uma só vez qualquer classe com qualquer número de argumentos construtores, para gerar qualquer tipo de instância:*

```
def factory(aClass, *args):               # tupla varargs
    return apply(aClass, args)            # Chama aClass.

class Spam:
    def doit(self, message):
        print message

class Person:
    def __init__(self, name, job):
        self.name = name
        self.job = job

object1 = factory(Spam)                   # Cria uma Spam.
object2 = factory(Person, "Guido", "guru")    # Cria uma Person.
```

* Na verdade, a função `apply` pode chamar qualquer objeto que possa ser chamado; isso inclui funções, classes e métodos. A função `factory` aqui pode executar qualquer objeto que possa ser chamado e não apenas uma classe (a despeito do nome do argumento).

Nesse código, definimos uma função geradora de objetos, chamada `factory`. Ela espera receber um objeto classe (qualquer classe servirá), junto com um ou mais argumentos do construtor da classe. A função usa `apply` para chamar a função e retornar uma instância.

O restante do exemplo simplesmente define duas classes e gera instâncias de ambas, passando-as para a função `factory`. E essa é a única função de fábrica que você precisará escrever em Python; ela funciona para qualquer classe e quaisquer argumentos construtores. Vale notar um possível aprimoramento: para suportar argumentos de palavra-chave nas chamadas do construtor, a função `factory` pode coletá-los com um argumento `**args` e passá-los como um terceiro argumento para `apply`:

```
def factory(aClass, *args, **kwargs)          # +kwargs dict
    return apply(aClass, args, kwargs)        # Chama aClass.
```

Agora você já deve saber que tudo é "objeto" no Python; até coisas como classes, que são apenas entrada de compilador em linguagens como a C++. Entretanto, conforme mencionado no início da Parte VI, no Python, apenas objetos derivados de classes são objetos da POO.

Por que fábricas?

Então, para que serve a função `factory` (além de nos dar uma desculpa para ilustrar objetos classe neste livro)? Infelizmente, é difícil mostrar aplicações desse padrão de projeto sem listar muito mais código do que temos espaço aqui para isso. Em geral, contudo, tal fábrica (`factory`) poderia permitir que código fosse isolado dos detalhes da construção de objetos configurados dinamicamente.

Por exemplo, lembre-se do exemplo `processor`, apresentado de forma abstrata no Capítulo 19 e, novamente, como um exemplo de composição tipo "tem um", neste capítulo. Ele aceitava objetos leitores e gravadores para processar fluxos de dados arbitrários. A versão original desse exemplo passava manualmente instâncias de classes especializadas, como `FileWriter` e `SocketReader`, para personalizar os fluxos de dados que estavam sendo processados. Posteriormente, passamos objetos arquivo, fluxo e formatador incorporados no código. Em um cenário mais dinâmico, os fluxos poderiam ser configurados por dispositivos externos, como arquivos de configuração ou GUIs.

Em tal mundo dinâmico, talvez não possamos incorporar a criação de objetos interface de fluxo no código de nosso script, mas poderíamos, em vez disso, criá-los no momento da execução, de acordo com o conteúdo de um arquivo de configuração. Por exemplo, o arquivo poderia simplesmente fornecer o nome da string de uma classe de fluxo a ser importada de um módulo, mais um argumento de chamada de construtor opcional. As funções ou código estilo fábrica podem ser úteis aqui, pois podemos buscar e passar classes que não são incorporadas antecipadamente no código do nosso programa. Na verdade, essas classes poderiam não existir quando escrevemos nosso código:

```
classname = ...parse from config file...
classarg = ...parse from config file...

import streamtypes                          # Código que pode ser personalizado
aclass = getattr(streamtypes, classname)    # Busca do módulo
reader = factory(aclass, classarg)          # ou aclass(classarg).
processor(reader, ...)
```

Aqui, novamente, a função interna `getattr` é usada para buscar um atributo de módulo, dado um nome de string (é como escrever `obj.attr`, mas `attr` é uma string). Como esse trecho de código presume um único argumento construtor, ele não precisa rigorosamente de `factory` nem de

Por outro lado, se qualificarmos a classe a obter como `doit`, receberemos de volta um objeto método *desvinculado*, o qual é simplesmente uma referência para o objeto função. Para chamar esse tipo de método, passe uma instância no argumento mais à esquerda:

```
object1 = Spam()
t = Spam.doit                          # Objeto método desvinculado
t(object1, 'howdy')                    # Passa a instância.
```

Por extensão, as mesmas regras se aplicam dentro do método de uma classe, se referenciarmos atributos `self` que fazem referência às funções presentes na classe. Um `self.método` é um objeto método vinculado, pois `self` é um objeto instância:

```
class Eggs:
    def m1(self, n):
        print n
    def m2(self):
        x = self.m1                    # Outro objeto método vinculado
        x(42)                          # Parece uma função simples

Eggs().m2()                            # Imprime 42
```

Na maioria das vezes, você chama métodos imediatamente após buscá-los com qualificação; portanto, nem sempre você nota os objetos método gerados pelo caminho. Mas se começar a escrever código que chama objetos genericamente, você precisará tomar o cuidado de tratar os métodos desvinculados de maneira especial – normalmente, eles exigem a passagem de um objeto instância explícito.*

STRINGS DE DOCUMENTAÇÃO REVISITADAS

O Capítulo 11 abordou as docstrings em detalhes, em nosso estudo das fontes e ferramentas de documentação. As docstrings são literais de string que aparecem no início de várias estruturas e são salvas automaticamente pelo Python em atributos de objeto `__doc__`. Isso funciona para arquivos de módulo, instruções `def` de função, classes e métodos. Agora que sabemos mais a respeito de classes e métodos, o arquivo *docstr.py* fornece um exemplo rápido, porém abrangente, que resume os lugares onde as docstrings podem aparecer em seu código; tudo pode ser blocos com três apóstrofos:

```
"I am: docstr.__doc__"

class spam:
    "I am: spam.__doc__ or docstr.spam.__doc__"

    def method(self, arg):
        "I am: spam.method.__doc__ or self.method.__doc__"
        pass

def func(args):
        "I am: docstr.func.__doc__"
        pass
```

A principal vantagem das strings de documentação é que elas persistem em tempo de execução; se um objeto tiver sido desenvolvido como uma string de documentação, você pode qualificá-lo para buscar sua documentação.

* Consulte a discussão posterior sobre a extensão de métodos *estáticos* e *de classe* no Python 2.2, para ver uma exceção opcional a essa regra. Assim como os métodos vinculados, ambos também podem ser mascarados como funções básicas, pois não esperam uma instância quando chamados.

CLASSES *VERSUS* MÓDULOS

Finalmente, vamos encerrar este capítulo comparando os assuntos das duas últimas partes deste livro – módulos e classes. Como ambos estão relacionados com espaços de nome, às vezes a distinção pode ser confusa. Em resumo:

Módulos

- São pacotes de dados/lógica
- São criados escrevendo-se arquivos em Python ou extensões em C
- São usados por meio de importação

Classes

- Implementam novos objetos
- São criadas por meio de instruções `class`
- São usadas por meio de chamadas
- Sempre residem dentro de um módulo

As classes também suportam recursos extras que os módulos não suportam, como sobrecarga de operador, geração de instância múltipla e herança. Embora ambos sejam espaços de nome, esperamos que agora você possa identificar que são coisas muito diferentes.

23

Tópicos Avançados das Classes

A Parte VI conclui nosso estudo da POO no Python, apresentando alguns assuntos mais avançados relacionados às classes, junto com os problemas e exercícios desta parte do livro. O estimulamos a fazer os exercícios para ajudar a cimentar as idéias que estudamos. Também sugerimos trabalhar ou estudar projetos de POO maiores em Python, como um complemento a este livro. Assim como acontece muito na computação, os benefícios da POO tendem a se tornar mais evidentes com a prática.

ESTENDENDO TIPOS INTERNOS

Além de implementar novos tipos de objetos, às vezes as classes são usadas para estender a funcionalidade dos tipos internos do Python, para suportar estruturas de dados mais exóticas. Por exemplo, para adicionar métodos de inserção e exclusão de filas, você pode desenvolver classes que encerram (incorporam) um objeto lista, e exportar métodos de inserção e exclusão que processam a lista de forma especial, como a técnica de delegação estudada no Capítulo 22. A partir do Python 2.2, você também pode usar herança para especializar tipos internos. As duas próximas seções mostram as duas técnicas em ação.

Estendendo tipos por meio de incorporação

Lembra-se daquelas funções de conjunto que escrevemos na Parte IV? Aqui está como elas se parecem, revividas como uma classe do Python. O exemplo a seguir, no arquivo *setwrapper.py*, implementa um novo tipo de objeto conjunto, movendo algumas das funções de conjunto para métodos e adicionando alguma sobrecarga de operador básica. De modo geral, essa classe apenas encerra uma lista do Python com operações de conjunto extras. Como se trata de uma classe, ela também suporta instâncias múltiplas e personalização por meio de herança em subclasses.

```
class Set:
    def __init__(self, value = []):        # Construtor
        self.data = []                     # Gerencia uma lista
        self.concat(value)
```

```
    def intersect(self, other):                    # other é qualquer seqüência.
        res = []                                   # self é o sujeito.
        for x in self.data:
            if x in other:                         # Seleciona itens comuns.
                res.append(x)
        return Set(res)                            # Retorna um novo Set.

    def union(self, other):                        # other é qualquer seqüência.
        res = self.data[:]                         # Cópia de minha lista
        for x in other:                            # Adiciona itens em other.
            if not x in res:
                res.append(x)
        return Set(res)

    def concat(self, value):                       # value: list, Set...
        for x in value:                            # Remove duplicatas
            if not x in self.data:
                self.data.append(x)

    def __len__(self):          return len(self.data)          # len(self)
    def __getitem__(self, key): return self.data[key]          # self[i]
    def __and__(self, other):   return self.intersect(other)   # self & other
    def __or__(self, other):    return self.union(other)       # self | other
    def __repr__(self):         return 'Set:' + `self.data`    # Imprime
```

Por meio da sobrecarga da indexação, a classe de conjunto freqüentemente pode ser mascarada como uma lista real. Como você vai interagir com essa classe e estendê-la em um exercício no final deste capítulo, não falaremos muito mais sobre esse código até o Apêndice B.

Estendendo tipos por meio de subclasses

A partir do Python 2.2, todos os tipos internos podem ser colocados em subclasses diretamente. As funções de conversão de tipo, como `list`, `str`, `dict` e `tuple`, tornaram-se nomes de tipo interno – embora seja transparente para seu script, uma chamada de conversão de tipo (por exemplo, `list('spam')`) agora é realmente a ativação do construtor de objetos de um tipo.

Essa mudança nos permite personalizar ou estender o comportamento de tipos internos com instruções `class` definidas pelo usuário: basta colocar os novos nomes de tipo em uma subclasse para personalizá-los. As instâncias de suas subclasses de tipo podem ser usadas em qualquer lugar onde o tipo interno original pode aparecer. Por exemplo, suponha que você tenha problemas para se acostumar com o fato de que o Python lista deslocamentos começando em 0, em vez de 1. Não se preocupe – você sempre pode desenvolver sua própria subclasse que personaliza esse comportamento básico das listas. O arquivo *typesubclass.py* mostra como:

```
# Coloca tipo/classe de lista interno em subclasse.
# Faz o mapeamento de 1..N para 0..N-1; chama de volta a versão interna.

class MyList(list):
    def __getitem__(self, offset):
        print '(indexing %s at %s)' % (self, offset)
        return list.__getitem__(self, offset - 1)

if __name__== '__main__':
    print list('abc')
    x = MyList('abc')               # __init__ herdado de list
    print x                         # __repr__ herdado de list
```

```
print x[1]                  # MyList.__getitem__
print x[3]                  # Personaliza método de superclasse de list

x.append('spam'); print x   # Atributos da superclasse de list
x.reverse(); print x
```

Nesse arquivo, a subclasse `MyList` estende apenas o método de indexação `__getitem__` da lista interna, para fazer o mapeamento de índices de 1 até N, de volta para o de 0 a N-1. Tudo que ele faz é decrementar o índice enviado e chamar de volta a versão de indexação da superclasse, mas é suficiente para resolver o problema:

```
% python typesubclass.py
['a', 'b', 'c']
['a', 'b', 'c']
(indexing ['a', 'b', 'c'] at 1)
a
(indexing ['a', 'b', 'c'] at 3)
c
['a', 'b', 'c', 'spam']
['spam', 'c', 'b', 'a']
```

Essa saída também inclui texto de rastreamento que a classe imprime na indexação. Se é uma boa idéia ou não alterar em geral a indexação dessa maneira, isso é outro problema – os usuários de sua classe `MyList` podem muito bem ser confundidos por esse desvio do comportamento de seqüência básico do Python. Contudo, o fato de você poder personalizar tipos internos dessa maneira pode ser uma ferramenta poderosa em geral.

Por exemplo, este padrão de desenvolvimento apresenta uma maneira alternativa de escrever conjuntos – como uma subclasse do tipo interno lista, em vez de uma classe independente que gerencia um objeto lista incorporado. A classe a seguir, desenvolvida no arquivo *setsubclass.py*, personaliza listas para adicionar apenas os métodos e operadores relacionados ao processamento de conjuntos. Como todos os outros comportamentos são herdados da superclasse interna `list`, isto se constitui em uma alternativa mais curta e mais simples:

```
class Set(list):
    def __init__(self, value = []):      # Construtor
        list.__init__([])                # Personaliza list
        self.concat(value)               # Copia padrões mutáveis

    def intersect(self, other):          # other é qualquer seqüência.
        res = []                         # self é o sujeito.
        for x in self:
            if x in other:               # Seleciona itens comuns.
                res.append(x)
        return Set(res)                  # Retorna um novo Set.

    def union(self, other):              # other é qualquer seqüência.
        res = Set(self)                  # Copia a mim e a minha lista.
        res.concat(other)
        return res

    def concat(self, value):             # value: list, Set...
        for x in value:                  # Remove duplicatas
        if not x in self:
            self.append(x)
```

```
    def __and__(self, other): return self.intersect(other)
    def __or__(self, other):  return self.union(other)
    def __repr__(self):       return 'Set:' + list.__repr__(self)

if __name__ == '__main__':
    x = Set([1,3,5,7])
    y = Set([2,1,4,5,6])
    print x, y, len(x)
    print x.intersect(y), y.union(x)
    print x & y, x | y
    x.reverse(); print x
```

Aqui está a saída do código de auto-teste no final do arquivo desse script. Como a colocação de tipos básicos em subclasses é um recurso avançado, omitiremos outros detalhes aqui, mas o convidamos a acompanhar esses resultados no código para estudar seu comportamento:

```
% python setsubclass.py
Set:[1, 3, 5, 7] Set:[2, 1, 4, 5, 6] 4
Set:[1, 5] Set:[2, 1, 4, 5, 6, 3, 7]
Set:[1, 5] Set:[1, 3, 5, 7, 2, 4, 6]
Set:[7, 5, 3, 1]
```

Existem maneiras mais eficientes de implementar conjuntos com dicionários no Python, os quais substituem as varreduras lineares nas implementações de conjunto que mostramos por operações de índice de dicionário (hashing) e, assim, são executados mais rapidamente. (Para ver mais detalhes, consulte o livro *Programming Python, Second Edition* [O'Reilly].) Se você estiver interessado em conjuntos, veja também o novo módulo `set` que foi adicionado no Python versão 2.3. Esse módulo fornece um objeto `set` e operações `set` como ferramentas internas. É divertido fazer experiências com conjuntos, mas eles não são mais rigorosamente exigidos a partir do Python 2.3.

Para outro exemplo de subclasse de tipo, veja a implementação do novo tipo `bool` no Python 2.3: conforme mencionado anteriormente, `bool` é uma subclasse de `int`, com duas instâncias, `True` e `False`, que se comportam como os inteiros 1 e 0, mas herdam métodos de representação de string personalizados que exibem seus nomes.

ATRIBUTOS DE CLASSE PSEUDO-PRIVADOS

Na Parte IV, aprendemos que todo nome atribuído no nível superior de um arquivo é exportado por um módulo. Por padrão, o mesmo vale para as classes – a ocultação de dados é uma convenção e os clientes podem buscar ou alterar qualquer atributo de classe ou de instância que quiserem. Na verdade, todos os atributos são "públicos" e "virtuais", nos termos de C++; todos eles são acessíveis em qualquer parte e todos são pesquisados dinamicamente no momento da execução.*

Isso ainda é verdade atualmente. Entretanto, o Python também inclui a noção de "desfiguração" (isto é, expansão) de nomes, para tornar locais alguns nomes nas classes. Às vezes isso é enganosamente chamado de atributos privados – na verdade, trata-se apenas de uma maneira de *tornar local* um nome na classe que o criou e não impede o acesso por código de fora da classe. Ou seja, esse recurso é principalmente destinado a evitar conflitos de espaço de nome em instâncias e não a restringir o acesso aos nomes em geral.

* Isso tende a assustar o pessoal de C++ desnecessariamente. No Python, é possível até alterar ou excluir completamente um método de classe em tempo de execução. Por outro lado, ninguém faz isso em programas práticos. Como uma linguagem de script, o Python é mais permissivo do que restritivo.

interno como `object`, a herança procura primeiro em `C` (à direita), antes de `A` (acima de `B`) – ela pesquisa `D`, `B`, `C` e depois `A` (e, neste caso, pára em `C`):

```
>>> class A(object): attr = 1          # Estilo novo
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B, C): pass                # Tenta C antes de A
>>> x = D()
>>> x.attr
2
```

Essa mudança na herança é baseada na suposição de que, se você mistura em `C`, mais abaixo na árvore, provavelmente pretende pegar seus atributos, em preferência aos de A. Isso também presume que `C` provavelmente *sempre* deve anular o atributo de `A` – isso acontece quando usada de forma independente, mas não quando misturada em um losango com classes clássicas. Você pode não saber que `C` pode ser misturada dessa forma, quando escreve isso.

Solução de conflitos explícita

É claro que o problema das suposições é que elas presumem coisas. Se esse desvio na ordem de pesquisa parece sutil demais para ser lembrado ou se você quer mais controle sobre o processo de pesquisa, sempre é possível forçar a seleção de um atributo em qualquer parte da árvore, atribuindo ou nomeando de outra forma o que você deseja, no lugar onde as classes são misturadas:

```
>>> class A: attr = 1                   # Clássica
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B, C): attr = C.attr        # Escolhe C, à direita.
>>> x = D()
>>> x.attr                              # Funciona como o estilo novo
2
```

Aqui, uma árvore de classes clássicas está simulando a ordem de pesquisa das classes de estilo novo. A atribuição ao atributo em `D` escolhe a versão em `C`, subvertendo com isso o caminho de pesquisa de herança normal (`D.attr` estará mais abaixo na árvore). Analogamente, as classes de estilo novo podem simular classes clássicas, escolhendo o atributo acima, no lugar onde as classes são misturadas:

```
>>> class A(object): attr = 1          # Estilo novo
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B, C): attr = B.attr       # Escolhe A.attr, acima.
>>> x = D()
>>> x.attr                             # Funciona como clássica
1
```

Se você quiser sempre solucionar conflitos como esse, pode ignorar a diferença na ordem de pesquisa e não contar com suposições a respeito do que quer dizer quando desenvolve suas classes. Naturalmente, os atributos que escolhemos dessa maneira também podem ser funções de método – os métodos são objetos normais que podem ser atribuídos:

```
>>> class A
...     def meth(s): print 'A.meth'
>>> class C(A):
...     def meth(s): print 'C.meth'
```

Métodos estáticos e de classe

É possível definir métodos dentro de uma classe, que podem ser chamados sem uma instância: os métodos *estáticos* funcionam quase como as funções sem instância simples dentro de uma classe e os métodos *de classe* recebem uma classe, em vez de uma instância. Funções internas especiais devem ser chamadas dentro da classe para ativar esses modos de método: `staticmethod` e `classmethod`. Como isso também é uma solução para um problema duradouro no Python, vamos apresentar essas chamadas posteriormente neste capítulo, na seção "Problemas das classes". Note que os novos métodos estáticos e de classe também funcionam para classes clássicas no Python versão 2.2.

Entradas de instância

Atribuindo-se uma lista de nomes de atributo de string ao atributo de classe especial `__slots__`, é possível para as classes de estilo novo limitarem o conjunto de atributos válidos que as instâncias da classe terão. Normalmente, esse atributo especial é configurado pela atribuição à variável `__slots__` no nível superior de uma instrução `class`. Apenas os nomes presentes na lista `__slots__` podem ser atribuídos como atributos de instância. Entretanto, assim como todos os nomes no Python, os nomes de atributo de instância ainda devem ser atribuídos antes de poderem ser referenciados, mesmo se listados em `__slots__`. Aqui está um exemplo para ilustrar:

```
>>> class limiter(object):
...     __slots__ = ['age', 'name', 'job']

>>> x = limiter()
>>> x.age                               # Precisa atribuir antes de usar
AttributeError: age

>>> x.age = 40
>>> x.age
40
>>> x.ape = 1000                        # Inválido: não está nas entradas
AttributeError: 'limiter' object has no attribute 'ape'
```

Esse recurso está previsto como uma maneira de capturar erros "tipográficos" (é detectada a atribuição a nomes de atributo inválidos que não estão em `__slots__`) e como um possível mecanismo de otimização no futuro. As entradas afetam a natureza dinâmica do Python, a qual prescreve que qualquer nome pode ser criado por atribuição. Elas também têm restrições e implicações adicionais que são complexas demais para discutirmos aqui (por exemplo, algumas instâncias com entradas podem não ter um dicionário de atributo `__dict__`). Para ver os detalhes, consulte os documentos do Python versão 2.2.

Propriedades de classe

Um mecanismo conhecido como propriedades fornece outra maneira para as classes de estilo novo definirem métodos chamados automaticamente para acesso ou atribuição a atributos de instância. Esse recurso é uma alternativa para muitos usos correntes dos métodos de sobrecarga `__getattr__` e `__setattr__`, estudados no Capítulo 21. As propriedades têm um efeito semelhante a esses dois métodos, mas acarretam uma chamada de método extra apenas para acessar nomes que exigem cálculo dinâmico. As propriedades (e as entradas) são baseadas em uma nova noção de descritores de atributo, que é avançada demais para abordarmos aqui.

Em resumo, as propriedades são um tipo de objeto atribuído a nomes de atributo de classe. Elas são geradas chamando-se o tipo interno `property`, com três métodos (rotinas de tratamen-

```
class Lister:
    def __repr__(self): ...
    def other(self): ...

class Super:
    def __repr__(self): ...
    def other(self): ...

class Sub(Lister, Super): # Obtém __repr__ de Lister listando-a primeiro, mas
                          # pega explicitamente a versão de Super de other.
    other = Super.other
    def __init__(self):
        ...

x = Sub()                 # A herança pesquisa Sub antes de Super/Lister.
```

Aqui, a atribuição a `other` dentro da classe `Sub` cria `Sub.other`—uma referência de volta para o objeto `Super.other`. Como está mais abaixo na árvore, `Sub.other` efetivamente oculta `List.other`, o atributo que a herança normalmente encontraria. Analogamente, se listássemos `Super` primeiro no cabeçalho de classe para escolher seu método `other`, precisaríamos então selecionar o método de `Lister`:

```
class Sub(Super, Lister):         # Obtém other de Super pela ordem.
    __repr__ = Lister.__repr__    # Escolhe Lister.__repr__ explicitamente.
```

A herança múltipla é uma ferramenta avançada. Mesmo que você tenha entendido o último parágrafo, ainda é uma boa idéia usá-la raramente e com cuidado. Caso contrário, o significado de um nome poderá depender da ordem em que as classes são misturadas em um subclasse arbitrariamente muito distante. Para outro exemplo da técnica mostrada aqui em ação, veja a discussão sobre solução de conflitos explícita, na seção "Classes de 'estilo novo' no Python 2.2", anteriormente neste capítulo.

Como regra geral, a herança múltipla funciona melhor quando suas classes de mistura são o mais auto-suficientes possível – como elas podem ser usadas em vários contextos, não devem fazer suposições sobre os nomes relacionados a outras classes em uma árvore. Além disso, o recurso dos atributos pseudo-privados que estudamos anteriormente pode ajudar, tornando locais os nomes que uma classe conta possuir e limitando os nomes que suas classes de mistura adicionam na mistura. No exemplo, se `Lister` só pretendesse exportar seu método personalizado `__repr__`, poderia chamar seu outro método de `__other` para evitar conflitos com outras classes.

Os atributos de função de classe são especiais: métodos estáticos

Este problema foi corrigido por um novo recurso opcional no Python 2.2, os métodos *estáticos* e *de classe*, mas o mantivemos aqui para os leitores que possuem versões mais antigas da linguagem e porque ele nos dá um bom motivo para apresentar o novo recurso avançado dos métodos estáticos e de classe.

Nas versões do Python anteriores a 2.2, as funções de método de classe nunca podem ser chamadas sem uma instância. (No Python 2.2 e posteriores, esse também é o comportamento padrão, mas pode ser modificado, se necessário.) No capítulo anterior, falamos sobre métodos *desvinculados*: quando buscamos uma função de método qualificando uma classe (em vez de uma instância), obtemos um objeto método desvinculado. Mesmo sendo definidos com uma instrução `def`, os objetos método desvinculado não são funções simples; eles não podem ser chamados sem uma instância.

Por exemplo, suponha que queiramos usar atributos de classe para contar quantas instâncias são geradas a partir de uma classe (o arquivo *spam.py*, mostrado a seguir). Lembre-se de que os atributos de classe são compartilhados por todas as instâncias, de modo que podemos armazenar o contador no próprio objeto classe:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print "Number of instances created: ", Spam.numInstances
```

Mas isso não funcionaria: o método `printNumInstances` ainda espera que uma instância seja passada, quando chamado, pois a função está associada a uma classe (mesmo não havendo nenhum argumento no cabeçalho de `def`):

```
>>> from spam import *
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: unbound method must be called with class instance 1st argument
```

Solução (antes da versão 2.2 e normalmente na 2.2)

Não espere isso: os métodos de instância desvinculados não são exatamente iguais às funções simples. Isso é principalmente uma questão de conhecimento, mas se você quiser chamar funções que acessam membros de classe sem uma instância, provavelmente o melhor conselho é apenas torná-los funções simples e não métodos de classe. Desse modo, uma instância não será esperada na chamada:

```
def printNumInstances():
    print "Number of instances created: ", Spam.numInstances

class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

>>> import spam
>>> a = spam.Spam()
>>> b = spam.Spam()
>>> c = spam.Spam()
>>> spam.printNumInstances()
Number of instances created: 3
>>> spam.Spam.numInstances
3
```

Também podemos fazer isso funcionar chamando por meio de uma instância, como sempre, embora isso possa ser inconveniente se o fato de criar uma instância alterar os dados da classe:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
```

```
        def printNumInstances(self):
            print "Number of instances created: ", Spam.numInstances

>>> from spam import Spam
>>> a, b, c = Spam(), Spam(), Spam()
>>> a.printNumInstances()
Number of instances created: 3
>>> b.printNumInstances()
Number of instances created: 3
>>> Spam().printNumInstances()
Number of instances created: 4
```

Alguns teóricos das linguagens de programação dizem que isso significa que o Python não tem métodos de classe, mas apenas métodos de instância. Suspeitamos que, na realidade, eles querem dizer que as classes do Python não funcionam da mesma forma que em algumas outras linguagens. Na realidade, o Python tem objetos método vinculado e desvinculado, com semântica bem definida. A qualificação de uma classe fornece a você um método desvinculado, que é um tipo especial de função. O Python não tem atributos de classe, mas as funções nas classes esperam um argumento de instância.

Além disso, como o Python já fornece *módulos* como ferramenta de particionamento de espaço de nome, normalmente não há necessidade de empacotar funções em classes, a não ser que elas implementem comportamento de objeto. Normalmente, funções simples dentro de módulos fazem a maior parte do que os métodos de classe sem instância poderiam fazer. Por exemplo, no primeiro exemplo de código desta seção, `printNumInstances` já está associada à classe, pois reside no mesmo módulo. A única funcionalidade perdida é que o nome da função tem um escopo mais amplo – o módulo inteiro, em vez da classe.

Métodos estáticos e de classe no Python 2.2

A partir do Python 2.2, você pode desenvolver classes com métodos estáticos e de classe, sendo que nenhum deles exige a presença de uma instância ao serem ativados. Para designar esses métodos, as classes chamam as funções internas `staticmethod` e `classmethod`, conforme sugerido na discussão anterior sobre as classes de estilo novo. Por exemplo:

```
class Multi:
    def imeth(self, x):             # Método de instância normal
        print self, x
    def smeth(x):                   # Estático: nenhuma instância passada
        print x
    def cmeth(cls, x):              # Classe: obtém a classe e não a instância
        print cls, x
    smeth = staticmethod(smeth)     # Torna smeth um método estático.
    cmeth = classmethod(cmeth)      # Torna cmeth um método de classe.
```

Observe como as duas últimas atribuições nesse código simplesmente *reatribuem* os nomes de método `smeth` e `cmeth`. Os atributos são criados e alterados por qualquer atribuição em uma instrução `class`; portanto, essas atribuições finais sobrescrevem as atribuições feitas anteriormente pelas instruções `def`.

Tecnicamente, o Python 2.2 suporta três tipos de métodos relacionados à classe: de instância, estáticos e de classe. Os *métodos de instância* são o caso normal (e padrão) que vimos neste livro. Com os métodos de instância, você sempre deve chamar o método com um objeto instância. Quando você chama por meio de uma instância, o Python passa a instância para o

primeiro argumento (mais à esquerda) automaticamente; quando chamado por meio da classe, você passa a instância manualmente:

```
>>> obj = Multi()                  # Cria uma instância
>>> obj.imeth(1)                   # Chamada normal, por meio de instância
<__main__.Multi instance...> 1
>>> Multi.imeth(obj, 2)            # Chamada normal, por meio de classe
<__main__.Multi instance...> 2
```

Em contraste, os *métodos estáticos* são chamados sem um argumento de instância. Seus nomes são locais no escopo da classe em que são definidos e podem ser pesquisados por herança. De modo geral, eles funcionam como funções simples que, por acaso, estão escritas dentro de uma classe:

```
>>> Multi.smeth(3)                 # Chamada estática, por meio de classe
3
>>> obj.smeth(4)                   # Chamada estática, por meio de instância
4
```

Os *métodos de classe* são semelhantes, mas o Python passa a classe (e não uma instância) automaticamente para o primeiro argumento (mais à esquerda) do método:

```
>>> Multi.cmeth(5)                 # Chamada de classe, por meio de classe
__main__.Multi 5
>>> obj.cmeth(6)                   # Chamada de classe, por meio de instância
__main__.Multi 6
```

Os métodos estáticos e de classe são recursos novos e avançados da linguagem. Eles têm funções altamente especializadas para as quais não temos espaço aqui para documentar. Os métodos estáticos são comumente usados em conjunto com atributos de classe para gerenciar informações que abrangem todas as instâncias geradas a partir da classe.

Por exemplo, para monitorar o número de instâncias geradas a partir de uma classe (como no exemplo anterior), você pode usar métodos estáticos para gerenciar um contador anexado como um atributo de classe. Como essa contagem não tem nada a ver com qualquer instância em particular, é conveniente tê-la para acessar métodos que a processam por meio de uma instância (especialmente porque criar uma instância para acessar o contador pode alterar a contagem). Além disso, a proximidade dos métodos estáticos com a classe fornece uma solução mais natural do que o desenvolvimento de funções orientadas a classes fora da classe. Aqui está o método estático equivalente do exemplo original desta seção:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print "Number of instances:", Spam.numInstances
    printNumInstances = staticmethod(printNumInstances)

>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()
Number of instances: 3
>>> a.printNumInstances()
Number of instances: 3
```

Comparada a simplesmente mover `printNumInstances` para fora da classe, conforme prescrito anteriormente, esta versão exige uma chamada de `staticmethod` extra, mas torna o nome da função local no escopo da classe e coloca o código da função mais perto de onde ele é usado (dentro da instrução `class`). Você mesmo deve julgar se isso é melhor ou não.

Métodos, classes e escopos aninhados

Este problema desapareceu no Python 2.2, com a introdução dos escopos de função aninhados, mas o mantivemos aqui por perspectiva histórica, para os leitores que trabalham com versões mais antigas da linguagem e porque ele demonstra o que acontece com as novas regras de escopo de função aninhado, quando uma instrução `class` é uma camada do aninhamento.

As classes introduzem um escopo local, assim como as funções, de modo que os mesmos tipos de comportamento de escopo podem acontecer no miolo de uma instrução `class`. Além disso, os métodos são funções mais aninhadas, de modo que os mesmos problemas se aplicam. A confusão parece ser particularmente comum quando classes são aninhadas.

No exemplo a seguir, no arquivo *nester.py*, a função `generate` retorna uma instância da classe aninhada `Spam`. Dentro de seu código, o nome de classe `Spam` é atribuído no escopo local da função `generate`. Mas dentro da função `method` da classe, o nome de classe `Spam` não é visível no Python antes da versão 2.2, onde `method` só tem acesso ao seu próprio escopo local, ao módulo circundante `generate` e aos nomes internos:

```
def generate():
    class Spam:
        count = 1
        def method(self):          # O nome Spam não é visível:
            print Spam.count       # não local(def),global(módulo), interno
    return Spam()

generate().method()

C:\python\examples> python nester.py
Traceback (innermost last):
  File "nester.py", line 8, in ?
generate().method()
  File "nester.py", line 5, in method
    print Spam.count               # Não local(def), global(módulo), interno
NameError: Spam
```

Como uma solução, migre para o Python 2.2 ou não aninhe código dessa maneira. Esse exemplo funciona no Python 2.2 e posteriores, pois os escopos locais de todas as instruções `def` de função são automaticamente visíveis para instruções `def` aninhadas, incluindo as instruções `def` de *método* aninhadas, como nesse exemplo.

Note que, mesmo na versão 2.2, as instruções `def` de método não podem ver o escopo local da *classe* envolvente, mas apenas o escopo local das instruções `def` envolventes. É por isso que os métodos devem passar pela instância de `self` ou pelo nome da classe, para referenciarem métodos e outros atributos definidos na instrução `class` envolvente. Por exemplo, o código no método deve usar `self.count` ou `Spam.count` e não apenas `count`.

Antes da versão 2.2, havia diversas maneiras de fazer o exemplo anterior funcionar. Uma das mais simples era mover o nome `Spam` para fora do escopo do módulo envolvente, com declarações globais. Como `method` enxerga nomes globais no módulo envolvente, as referências funcionam:

```
def generate():
    global Spam                     # Força Spam no escopo do módulo.
    class Spam:
        count = 1
        def method(self):
            print Spam.count        # Funciona: em global (módulo envolvente)
    return Spam()

generate().method()                 # Imprime 1
```

Talvez melhor ainda, também podemos reestruturar o código de modo que `Spam` seja definida no nível superior do módulo, graças ao seu nível de aninhamento, em vez de usar declarações globais. Tanto a função aninhada `method` quanto a função de nível superior `generate` encontram `Spam` em seus escopos globais:

```
def generate():
    return Spam()
class Spam:                         # Define no nível superior do módulo.
    count = 1
    def method(self):
        print Spam.count            # Funciona: em global (módulo envolvente)

generate().method()
```

Na verdade, isso é o que prescrevemos para todas as versões do Python – em geral, seu código tende a ser mais simples se você evita o aninhamento de classes e funções.

Se quiser ser mais complicado e difícil, você também pode se desfazer completamente da referência de `Spam` em `method`, usando o atributo especial `__class__`, que retorna o objeto classe de uma instância:

```
def generate():
    class Spam:
        count = 1
        def method(self):
            print self.__class__.count    # Funciona: qualifica para
                                          # obter a classe
    return Spam()

generate().method()
```

Super-encerramentite

Às vezes, o potencial de abstração da POO pode ser abusado a ponto de tornar o código difícil de entender. Se suas classes são dispostas em camadas com profundidade demasiada, isso pode tornar o código obscuro. Talvez você tenha que pesquisar muitas classes para descobrir o que uma operação faz. Por exemplo, uma vez, um dos autores deste livro trabalhou em um programa em C++ com milhares de classes (algumas geradas por máquina) e até 15 níveis de herança. Decifrar uma chamada de método em um sistema tão complexo era freqüentemente uma tarefa monumental. Várias classes tinham que ser consultadas, mesmo para a mais básica das operações.

A regra mais geral se aplica aqui também: não torne as coisas complicadas, a não ser que elas realmente precisem ser. Encerrar seu código em múltiplas camadas de classes, a ponto de torná-lo incompreensível, é sempre uma má idéia. A abstração é a base do polimorfsmo e do encapsulamento, e pode ser uma ferramenta muito eficiente, quando bem utilizada. Mas você simplificará a depuração e ajudará na manutenção, se tornar suas interfaces de classe

Por que isto é relevante: POO pelos mestres

Quando eu (Mark) ensino Python, invariavelmente, lá pelo meio da aula, as pessoas que já usaram POO estão acompanhando intensamente; os olhos das pessoas que não usaram estão começando a embaçar (ou elas já estão cochilando). O objetivo por trás da tecnologia simplesmente não está aparente.

Em um livro como este, temos o luxo de adicionar material de visão geral, como o novo "Panorama geral" no Capítulo 19, e você provavelmente deve rever aquela seção, se estiver começando a achar que a POO é apenas alguma superstição da ciência da computação.

Entretanto, nas aulas reais, para ajudar os iniciantes a embarcar no assunto (e despertarem), eu paro e pergunto aos especialistas na platéia por que eles usam POO afinal. As respostas que eles têm dado podem ajudar a jogar alguma luz no objetivo da POO, caso você seja iniciante no assunto.

Então, aqui está (com apenas alguns enfeites) a maioria dos motivos comuns para se usar POO, conforme citado por meus alunos com o passar dos anos:

Reutilização de código

Este é fácil (e é o principal motivo para se usar POO). Suportando herança, as classes permitem que você programe por meio de personalização, em vez de começar cada projeto desde o início.

Encapsulamento

Encerrando os detalhes da implementação atrás de interfaces de objeto, os usuários de uma classe ficam isolados das alterações feitas no código.

Estrutura

As classes fornecem um novo escopo local, o que minimiza os conflitos de nome. Elas também fornecem um lugar natural para escrever e procurar código de implementação e gerenciar estado de objeto.

Manutenção

Graças à estrutura e ao suporte para reutilização de código das classes, normalmente há apenas uma cópia de código a ser alterada.

Consistência

As classes e a herança permitem que você implemente interfaces comuns e, assim, aparência e comportamento comuns em seu código. Isso facilita a depuração, a compreensão e a manutenção.

Polimorfismo

Esta é mais uma propriedade da POO do que um motivo, mas, suportando generalidade de código, o polimorfismo torna o código mais flexível e amplamente aplicável, tornando-o com isso mais reutilizável.

E, é claro, o motivo número um que os alunos deram para usar POO: fica bem no currículo.

Finalmente, lembre-se do que dissemos no início da Parte VI: você não apreciará a POO totalmente, até tê-la usado um pouco. Pegue um projeto, estude exemplos maiores, faça os exercícios – o que for necessário para que você comece a usar código OO, vale o esforço.

e. Que tipo de objeto operações como + e fracionamento devem retornar? E quanto à indexação?
f. Se você estiver trabalhando com uma versão mais recente do Python (2.2 ou posterior), pode implementar esse tipo de classe `wrapper` incorporando uma lista real

Note que Lunch precisa passar Employee para Customer ou passar a si mesma para Customer, para permitir que Customer chame métodos de Employee.

Experimente suas classes interativamente, importando a classe Lunch, chamando seu método order para executar uma interação e depois chamando seu método result para verificar se Customer recebeu o que pediu. Se preferir, você também pode simplesmente desenvolver casos de teste como código de auto-teste no arquivo onde suas classes são definidas, usando o truque do módulo __name__ do Capítulo 18. Nessa simulação, Customer é o agente ativo. Como suas classes mudariam se, em vez disso, Employee fosse o objeto que iniciasse a interação cliente/funcionário?

8. *Hierarquia de animais do zoológico.* Considere a árvore de classes mostrada na Figura 23-1. Escreva um conjunto de seis instruções class para modelar essa taxonomia com herança do Python. Em seguida, adicione em cada uma de suas classes um método speak que imprima uma mensagem exclusiva e um método reply, em sua superclasse de nível superior Animal, que simplesmente chame self.speak para ativar a impressora de mensagem específica da categoria em uma subclasse abaixo (isso provocará uma pesquisa de herança independente de self). Finalmente, remova o método speak de sua classe Hacker, para que ela escolha o padrão que está acima dela. Quando você tiver terminado, suas classes deverão funcionar como segue:

```
% python
>>> from zoo import Cat, Hacker
>>> spot = Cat()
>>> spot.reply()              # Animal.reply; chama Cat.speak
meow
>>> data = Hacker()           # Animal.reply; chama Primate.speak
>>> data.reply()
Hello world!
```

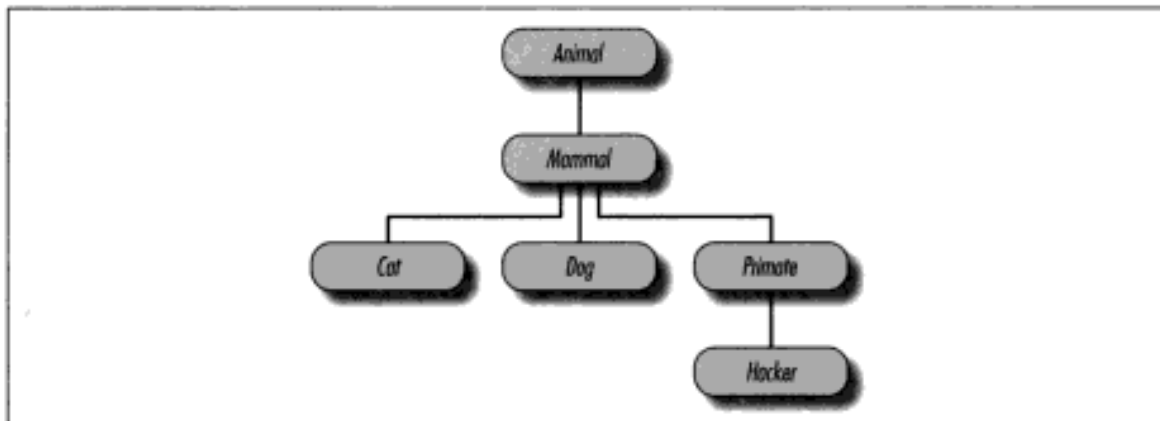



Figura 23-1 Uma hierarquia de zoológico.

9. *O esboço do papagaio morto.* Considere a estrutura de incorporação de objeto capturada na Figura 23-2. Escreva um conjunto de classes Python para implementar essa estrutura com composição. Desenvolva seu objeto Scene para definir um método action e incorpore instâncias das classes Customer, Clerk e Parrot – todas as três devem definir um método line que imprima uma mensagem exclusiva. Os objetos incorporados podem herdar de uma superclasse comum que define line e simplesmente fornecer o texto da mensagem ou definirem line eles mesmos. No final, suas classes devem operar como segue:

```
% python
>>> import parrot
>>> parrot.Scene().action()        # Ativa objetos aninhados.
customer: "that's one ex-bird!"
clerk: "no it isn't..."
parrot: None
```

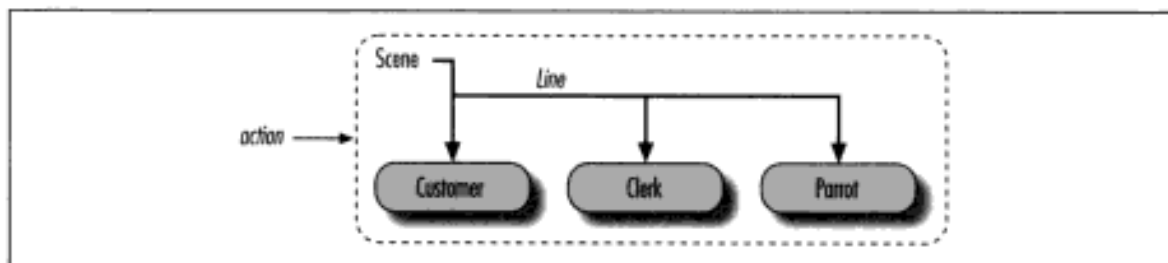



Figura 23-2 Uma composição de cena.

VII

Exceções e Ferramentas

Na Parte VII, estudaremos as exceções, que são eventos de software gerados pelo Python no caso de erros, ou em seu programa por demanda. As exceções podem ser capturadas e ignoradas ou podem passar e terminar um programa com uma mensagem de erro padrão. Por isso, a Parte VII também está relacionada com a história da depuração no Python. Conforme veremos, embora as mensagens de erro padrão freqüentemente sejam suficientes para analisar um problema, o tratamento de exceções é uma ferramenta leve que oferece mais controle.

A Parte VII também inicia a transição dos assuntos básicos da linguagem para os assuntos das ferramentas periféricas. Rigorosamente falando, as exceções representam o último assunto básico da linguagem que conheceremos no livro. Após a Parte VII, lidaremos com ferramentas da biblioteca padrão e de domínio público, além da linguagem em si. Concluiremos esta parte com uma visão geral das ferramentas que são úteis no desenvolvimento de aplicativos maiores em Python.

24

Fundamentos das Exceções

A Parte VII trata das *exceções*, que são eventos que podem modificar o fluxo de controle em um programa. No Python, as exceções são lançadas automaticamente no caso de erros e podem ser lançadas e interceptadas pelo seu código. Elas são processadas pelas três instruções que estudaremos nesta parte, a primeira das quais possui duas variações:

`try/except`
: Captura e se recupera de exceções lançadas pelo Python ou por você.

`try/finally`
: Executa ações de limpeza, ocorram exceções ou não.

`raise`
: Lança uma exceção manualmente em seu código.

`assert`
: Lança uma exceção condicionalmente em seu código.

Com algumas exceções (trocadilho intencional), veremos que o tratamento de exceções é simples no Python, pois é integrado na própria linguagem como outra ferramenta de alto nível.

POR QUE USAR EXCEÇÕES?

Em resumo, as exceções nos permitem pular trechos arbitrariamente grandes de um programa. Considere o robô pizzaiolo sobre o qual falamos anteriormente no livro. Suponha que levássemos a idéia a sério e realmente construíssemos tal máquina. Para fazer uma pizza, nosso autômato da culinária precisaria executar um plano, o qual implementaríamos como um programa em Python. Ele pegaria um pedido, prepararia a massa, adicionaria os ingredientes, assaria etc.

Agora, suponha que algo desse errado durante a etapa "assar". Talvez o forno não estivesse funcionando ou talvez nosso robô calculasse mal o seu alvo e se queimasse espontaneamente. Claramente, nós queremos estar aptos para pular para o código que trata de tais estados rapidamente. Como não temos nenhuma esperança de terminar a tarefa da pizza em tais casos incomuns, podemos abandonar o plano inteiramente.

É exatamente isso que as exceções permitem fazer: você pode pular para uma rotina de tratamento de exceção em um único passo, abandonando todas as chamadas de função suspensas. Elas são uma espécie de "super-goto"* estruturado. Uma rotina de tratamento de exceção (a instrução `try`) deixa para trás um marcador e executa algum código. Em algum lugar mais adiante no programa, é lançada uma exceção que faz o Python voltar para o marcador imediatamente, sem retomar nenhuma das funções ativas que foram chamadas desde que o marcador foi deixado. O código da rotina de tratamento de exceção pode responder à exceção lançada conforme for apropriado (chamando o corpo de bombeiros, por exemplo). Além disso, como o Python pula imediatamente para a instrução da rotina de tratamento, normalmente não há necessidade de verificar códigos de status após cada chamada para uma função que possivelmente poderia falhar.

Funções da exceção

Nos programas em Python, as exceções são normalmente usadas para uma variedade de propósitos. Aqui estão algumas de suas funções mais comuns:

Tratamento de erros

O Python lança exceções ao detectar erros nos programas em tempo de execução. Você pode capturar e responder aos erros em seu código ou ignorar a exceção. Se o erro for ignorado, o comportamento padrão do tratamento de exceção entra em ação – ele interrompe o programa e imprime uma mensagem de erro. Se você não quiser esse comportamento padrão, escreva uma instrução `try` para capturar e se recuperar da exceção – o Python pula para sua rotina de tratamento `try` quando o erro é detectado e seu programa retoma a execução após a rotina `try`.

Notificação de eventos

As exceções também podem sinalizar uma condição válida, sem ter de passar flags de resultado em um programa ou testá-los explicitamente. Por exemplo, uma rotina de pesquisa poderia lançar uma exceção em caso de falha, em vez de retornar um código de resultado de valor inteiro (e esperar que o código nunca seja um resultado válido).

Tratamento de casos especiais

Às vezes, uma condição pode ocorrer tão raramente que é difícil justificar uma complicação em seu código para tratar dela. Freqüentemente, você pode eliminar código de caso especial tratando dos casos incomuns em rotinas de tratamento de exceção.

Ações de finalização

Conforme veremos, a instrução `try/finally` nos permite garantir que operações exigidas no momento do fechamento sejam executadas, independente da presença ou da ausência de exceções em seu programa.

Fluxos de controle incomuns

Finalmente, como as exceções são uma espécie de "goto" de alto nível, você pode usálas como base para implementar fluxos de controle exóticos. Por exemplo, embora a

* Se você já usou a linguagem C, pode estar interessado em saber que as exceções do Python são quase semelhantes ao par de funções padrão `setjmp/longjmp` daquela linguagem. A instrução `try` age de forma muito parecida com `setjmp` e `raise` funciona como `longjmp`. Mas, no Python, as exceções são baseadas em objetos e são uma parte padrão do modelo de execução.

reversão não faça parte da linguagem em si, ela pode ser implementada no Python com exceções e alguma lógica de suporte para desfazer atribuições.*

Veremos alguns usos típicos em ação posteriormente, nesta parte do livro. Primeiramente, vamos começar examinando as ferramentas de processamento de exceção do Python.

TRATAMENTO DE EXCEÇÕES: A HISTÓRIA BREVE

Comparadas aos outros assuntos básicos da linguagem que já conhecemos, as exceções são uma ferramenta bastante leve no Python. Como elas são muito simples, vamos diretamente a um exemplo inicial. Suponha que você tenha desenvolvido a seguinte função:

```
>>> def fetcher(obj, index):
...     return obj[index]
```

Não há muita coisa nessa função – ela simplesmente indexa um objeto em um índice passado. Na operação normal, ela retorna o resultado dos índices válidos:

```
>>> x = 'spam'
>>> fetcher(x, 3)            # É como x[3]
'm'
```

Entretanto, se você pedir a essa função para que indexe após o final de sua string, lançará uma exceção quando a função tentar executar `obj[index]`. O Python detecta a indexação de seqüência fora do limite e relata isso *lançando* (disparando) a exceção interna `IndexError`:

```
>>> fetcher(x, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

Tecnicamente, como essa exceção não é capturada por seu código, ela atinge o nível superior do programa e ativa a *rotina de tratamento de exceção padrão* – a qual simplesmente imprime a mensagem de erro padrão. Neste ponto do livro, você provavelmente já viu várias mensagens de erro padrão. Elas incluem a exceção que foi lançada, junto com um *rastreamento de pilha* – uma lista das linhas e funções que estavam ativas quando a exceção ocorreu. Ao se desenvolver interativamente, o arquivo é apenas "stdin" (fluxo de entrada padrão) ou "pyshell" (no IDLE); portanto, os números de linha de arquivo não são muito significativos aqui.

Em um programa mais realista, executado fora do prompt interativo, a rotina de tratamento padrão na parte superior também *termina* o programa imediatamente. Esse curso de ação faz sentido para scripts simples; freqüentemente, os erros devem ser fatais e o melhor que você pode fazer é inspecionar a mensagem de erro padrão. Às vezes, contudo, não é isso que você quer. Os programas em servidores, por exemplo, normalmente precisam permanecer ativos, mesmo após a ocorrência de erros internos. Se você não quer o comportamento de exceção padrão, encerre a chamada em uma instrução `try` para capturar a exceção por conta própria:

```
>>> try:
...     fetcher(x, 4)
```

* A reversão verdadeira é um assunto avançado que não faz parte da linguagem Python (mesmo com a adição de funções geradoras na versão 2.2); portanto, não falaremos mais nada sobre isso aqui. A grosso modo, a reversão desfaz toda computação realizada antes do salto. As exceções do Python não fazem isso (por exemplo, as variáveis atribuídas entre o momento em que se entra em uma instrução `try` e o momento em que uma exceção é lançada não são reconfiguradas com seus valores anteriores). Consulte um livro sobre inteligência artificial ou sobre as linguagens de programação Prolog ou icon, caso esteja curioso.

```
... except IndexError:
...     print 'got exception'
...
got exception
>>>
```

Agora, o Python pula automaticamente para sua *rotina de tratamento* (o bloco sob a cláusula `except` que nomeia a exceção lançada), quando a exceção é lançada, enquanto o bloco `try` é executado. Ao se trabalhar interativamente, dessa forma, após a cláusula `except` ser executada, acabamos voltando ao prompt do Python. Em um programa mais realista, as instruções `try` não apenas *capturam* as exceções, como também se *recuperam* delas:

```
>>> def catcher():
...     try:
...         fetcher(x, 4)
...     except IndexError:
...         print 'got exception'
...     print 'continuing'
...
>>> catcher()
got exception
continuing
>>>
```

Desta vez, depois que a exceção é capturada e tratada, o programa retoma a execução após a instrução `try` inteira que a capturou – que é o motivo pelo qual recebemos a mensagem "continuing" aqui. Você não verá a mensagem de erro padrão e seu programa continuará em seu caminho normalmente.

As exceções podem ser lançadas pelo Python e por você, e podem ser capturadas ou não. Para lançar uma exceção manualmente, basta executar uma instrução `raise` (ou `assert`). As exceções definidas pelo usuário são capturadas da mesma maneira que as internas:

```
>>> bad = 'bad'
>>> try:
...     raise bad
... except bad:
...     print 'got bad'
...
got bad
```

Se não forem capturadas, as exceções definidas pelo usuário atingirão a rotina de tratamento de exceção de nível superior padrão e terminarão seu programa com uma mensagem de erro padrão. Nesse caso, a mensagem padrão incluirá o texto da string usada para identificar a exceção:

```
>>> raise bad
Traceback(most recent call last):
  File "<pyshell#18>", line 1, in ?
    raise bad
bad
```

Em outros casos, a mensagem de erro pode incluir texto fornecido por *classes* usadas para identificar exceções. Conforme veremos no próximo capítulo, as exceções baseadas em classe permitem que scripts construam categorias de exceção (e outras coisas):

```
>>> class Bad: pass
...
```

Nessa instrução, o bloco sob o cabeçalho `try` representa a *ação principal* da instrução – o código que você está tentando executar. As cláusulas `except` definem *rotinas de tratamento* para as exceções lançadas durante o bloco `try` e a cláusula `else` (se for escrita) fornece uma rotina de tratamento a ser executada caso *nenhuma* exceção ocorra. A entrada `<dados>` aqui está relacionada com um recurso das instruções `raise` que discutiremos posteriormente neste capítulo.

Aqui está como as instruções `try` funcionam. Quando uma instrução `try` é iniciada, o Python marca o contexto do programa corrente, para que possa voltar caso ocorra uma exceção. As instruções aninhadas sob o cabeçalho `try` são executadas primeiro. O que acontece em seguida depende de exceções serem lançadas ou não, enquanto as instruções do bloco `try` estão executando:

- Se ocorre uma exceção enquanto as instruções do bloco estão executando, o Python volta para a instrução `try` e executa as instruções que estão sob a primeira cláusula `except` correspondente à exceção lançada. O controle continua após a instrução `try` inteira, depois que o bloco `except` é executado (a não ser que o bloco `except` lance outra exceção).
- Se uma exceção acontece no bloco `try` e não há *nenhuma* cláusula `except` correspondente, a exceção é propagada para cima, até uma instrução `try` em que se entrou anteriormente no programa ou para o nível superior do processo (o que faz o Python eliminar o programa e imprimir uma mensagem de erro padrão).
- Se nenhuma exceção ocorre enquanto as instruções sob o cabeçalho `try` são executadas, o Python executa as instruções que estão sob a linha `else` (se estiver presente) e, então, o controle retoma após a instrução `try` inteira.

Em outras palavras, as cláusulas `except` capturam as exceções que podem ocorrer enquanto o bloco `try` está em execução.

As cláusulas `except` são rotinas de tratamento de exceção focalizadas – elas capturam exceções que ocorrem somente dentro das instruções que estão no bloco `try` associado. Entretanto, como as instruções do bloco `try` podem chamar funções escritas em qualquer parte de um programa, a origem de uma exceção pode estar fora da instrução `try` em si. Mais informações sobre isso aparecerão quando explorarmos o aninhamento de `try`, no Capítulo 26.

Cláusulas da instrução try

Quando você escreve instruções `try`, uma variedade de cláusulas pode aparecer após o bloco da instrução `try`. A Tabela 24-1 resume todas as formas possíveis e você deve usar pelo menos uma. Já conhecemos algumas delas – as cláusulas `except` capturam exceções, `finally` executa ao sair etc. Sintaticamente, pode haver qualquer número de cláusulas `except`, mas apenas uma cláusula `else`. Além disso, a cláusula `finally` deve aparecer sozinha (sem `else` ou `except`); trata-se, na verdade, de uma instrução diferente.

Tabela 24-1 Formas de cláusula da instrução `try`

Forma de cláusula	Interpretação
`except:`	Captura todos os (outros) tipos de exceção.
`except nome:`	Captura apenas uma exceção específica.
`except nome, valor:`	Captura a exceção e seus dados extras.
`except (nome1, nome2):`	Captura qualquer uma das exceções listadas.
`except (nome1, nome2), valor`	Captura qualquer uma e obtém os dados extras.

Tabela 24-1 Formas de cláusula da instrução `try` *(continuação)*

Forma de cláusula	Interpretação
`else:`	Executa o bloco se nenhuma exceção for lançada.
`finally:`	Sempre executa o bloco.

Vamos explorar as entradas com a parte extra `valor`, quando conhecermos a instrução `raise`. A primeira e a quarta entradas da Tabela 24-1 são novas aqui:

- As cláusulas `except` que não listam nenhum nome de exceção capturam *todas* as exceções não listadas anteriormente na instrução `try` (`except:`).
- As cláusulas `except` que listam um conjunto de exceções entre parênteses capturam *qualquer* uma das exceções listadas (`except (e1,e2,e3)`).

Como o Python procura uma correspondência dentro de determinada instrução `try` inspecionando as cláusulas `except` de cima para baixo, a versão com parênteses é como listar cada exceção em sua própria cláusula `except`, mas o miolo da instrução precisa ser escrito apenas uma vez. Aqui está um exemplo de múltiplas cláusulas `except` em funcionamento, o qual demonstra exatamente como podem ser suas rotinas de tratamento específicas:

```
try:
    action()
except NameError:
    ...
except IndexError
    ...
except KeyError
    ...
except (AttributeError, TypeError, SyntaxError):
    ...
else:
    ...
```

Nesse exemplo, quando uma exceção é lançada enquanto a chamada para a função `action` está executando, o Python retorna para a instrução `try` e procura a primeira cláusula `except` que nomeia a exceção lançada. Ela inspeciona as cláusulas `except` de cima para baixo e da esquerda para a direita, e executa as instruções que estão sob a primeira que corresponda. Se nenhuma corresponder, a exceção é propagada para depois dessa instrução `try`.

Note que a cláusula `else` é executada apenas quando *nenhuma* exceção ocorreu em `action` e não para as outras exceções lançadas. Se você quiser uma cláusula geral para *capturar tudo*, uma instrução `except` *vazia* resolverá:

```
try:
    action()
except NameError:
    ...                         # Trata de NameError.
except IndexError
    ...                         # Trata de IndexError.
except:
    ...                         # Trata de todas as outras exceções.
else:
    ...                         # Trata do caso em que não há nenhuma exceção.
```

A cláusula `except` vazia é uma espécie de recurso curinga – como captura tudo, ela permite que suas rotinas de tratamento sejam tão gerais ou específicas quanto você queira. Em alguns cenários, essa forma pode ser mais conveniente do que listar todas as exceções possíveis em uma instrução `try`. Por exemplo, o código a seguir captura tudo sem listar nada:

```
try:
    action()
except:
    ...                 # Captura todas as exceções possíveis.
```

As cláusulas `except` vazias também acarretam alguns problemas de projeto. Embora sejam convenientes, elas também podem capturar exceções de sistema inesperadas, não relacionadas com seu código, e podem interceptar inadvertidamente exceções destinadas a outra rotina de tratamento. Por exemplo, no Python, até as chamadas de saída do sistema lançam exceções e, normalmente, você quer que elas passem. Vamos rever isso como um problema no final da Parte VII. Por enquanto, diremos apenas: use com cuidado.

A cláusula try/else

À primeira vista, o objetivo da cláusula `else` nem sempre é óbvio. Sem ela, contudo, não há como identificar (sem configurar e verificar flags booleanos) se passamos de uma instrução `try` porque nenhuma exceção ocorreu ou porque ocorreu uma exceção e ela foi tratada:

```
try:
    ...executa código...
except IndexError:
    ...trata da exceção...
# Chegamos aqui porque a instrução try falhou ou não?
```

Assim como as cláusulas `else` em loops, aqui ela fornece sintaxe que torna esse caso óbvio e inequívoco:

```
try:
    ...executa código...
except IndexError:
    ...trata da exceção...
else:
    ...nenhuma exceção ocorreu...
```

Você *quase* pode simular uma cláusula `else` movendo seu código para o final do bloco `try`:

```
try:
    ...executa código...
    ...nenhuma exceção ocorreu...
except IndexError:
    ...trata da exceção...
```

Contudo, isso pode levar a classificações de exceção incorretas. Se a ação de "nenhuma exceção ocorreu" disparar `IndexError`, ela será registrada como uma falha do bloco `try` e, com isso, ativará erroneamente a rotina de tratamento de exceção que está abaixo da instrução `try` (sutil, mas verdadeiro!). Usando, em vez disso, uma cláusula `else` explícita, você torna a lógica mais evidente e garante que as rotinas de tratamento de `except` só sejam executadas para falhas reais no código que está encerrando no bloco `try`, e não para falhas na ação do caso `else`.

Exemplo: comportamento padrão

Como o fluxo de controle por um programa é mais fácil de capturar no Python do que em português, vamos executar alguns exemplos que ilustram melhor os fundamentos da exceção. As exceções não capturadas por instruções `try` atingem o nível superior de um processo do Python e executam a lógica de tratamento de exceção padrão da linguagem. O Python termina a execução do programa e imprime uma mensagem de erro padrão. Por exemplo, executar o módulo a seguir, *bad.py*, gera uma exceção de divisão por zero:

```
def gobad(x, y):
    return x / y

def gosouth(x):
    print gobad(x, 0)

gosouth(1)
```

Como o programa ignora a exceção que lança, o Python o eliminará e imprimirá uma mensagem – desta vez, com informações úteis sobre o arquivo e sobre o número da linha:*

```
% python bad.py
Traceback (most recent call last):
  File "bad.py", line 7, in ?
    gosouth(1)
  File "bad.py", line 5, in gosouth
    print gobad(x, 0)
  File "bad.py", line 2, in gobad
    return x / y
ZeroDivisionError: integer division or modulo by zero
```

Quando ocorre uma exceção não capturada, o Python termina o programa e imprime um rastreamento de pilha, o nome e todos os dados extras da exceção que foi lançada. O rastreamento de pilha lista todas as linhas que estavam ativas quando a exceção ocorreu, da mais antiga para a mais recente. Por exemplo, você pode ver que a divisão errada acontece na última entrada do rastreamento – a linha 2 do arquivo *bad.py*, uma instrução `return`.

Como o Python detecta e relata todos os erros em tempo de execução, lançando exceções, estas estão intimamente ligadas à idéia de tratamento de erros em geral. Por exemplo, se você trabalhou nos exemplos, sem dúvida viu uma ou duas exceções pelo caminho – até erros tipográficos normalmente geram uma exceção `SyntaxeError` ou outra, quando um arquivo é importado ou executado (que é quando o compilador é executado). Por padrão, você recebe uma tela de erros útil, como a anterior, a qual ajuda a rastrear o problema.

Freqüentemente, essa mensagem de erro padrão é tudo que você precisa para resolver um problema em seu código. Para tarefas de depuração mais pesadas, você pode capturar exceções com instruções `try` ou usar as ferramentas de depuração que apresentaremos no Capítulo 26.

Exemplo: capturando exceções internas

Freqüentemente, o tratamento de exceções padrão do Python é exatamente o que você quer – especialmente para código em arquivos de script de nível superior, um erro geralmente deve

* Devemos mencionar que o texto das mensagens de erro e os rastreamentos de pilha tendem a variar ligeiramente com o passar do tempo. Não se preocupe se suas mensagens de erro não forem exatamente iguais às nossas.

terminar seu programa imediatamente. Para muitos programas, não há necessidade de ser mais específico quanto aos erros existentes em seu código.

Contudo, às vezes você desejará, em vez disso, capturar erros e se recuperar deles. Se você não quer que seu programa termine quando uma exceção for lançada pelo Python, basta capturá-la, encerrando a lógica do programa em um bloco `try`. Por exemplo, o código a seguir captura e se recupera da exceção `TypeError` que o Python lança imediatamente, quando tentamos concatenar uma lista e uma string (o operador + exige o mesmo tipo de seqüência nos dois lados):

```
def kaboom(x, y):
    print x + y                    # Lança TypeError.

try:
    kaboom([0,1,2], "spam")
except TypeError:                  # Captura e se recupera aqui.
    print 'Hello world!'
print 'resuming here'              # Sempre executa este código ao sair.
```

Quando uma exceção ocorre na função `kaboom`, o controle pula para a cláusula `except` da instrução `try`, a qual imprime uma mensagem. Como uma exceção está "morta" após ter sido capturada dessa forma, o programa continua após o bloco `try` inteiro, em vez de ser finalizado pelo Python. Na verdade, seu código processa e elimina o erro.

Note que, uma vez capturado o erro, o controle retoma no lugar onde você o capturou, após a instrução `try`; não há nenhuma maneira direta de voltar para o lugar onde a exceção ocorreu (função `kaboom`). De certo modo, isso torna as exceções mais parecidas com saltos simples do que com chamadas de função – não há nenhuma maneira de retornar para o código que disparou o erro.

A INSTRUÇÃO try/finally

O outro tipo de instrução `try` é uma especialização e está relacionado com ações de finalização. Se uma cláusula `finally` for usada em uma instrução `try`, seu bloco de instruções sempre será executado pelo Python "ao sair", tenha ocorrido ou não uma exceção enquanto o bloco `try` estava sendo executado. Sua forma geral é:

```
try:
    <instruções>                   # Executa esta ação primeiro.
finally:
    <instruções>                   # Sempre executa este código ao sair.
```

Aqui está como esta variante funciona. O Python começa executando primeiro o bloco de instruções associado à linha de cabeçalho de `try`. O comportamento restante dessa instrução depende de uma exceção ocorrer ou não durante o bloco `try`:

- Se nenhuma exceção ocorre enquanto o bloco `try` está em execução, o Python pula para executar o bloco `finally` e depois continua a execução após a instrução `try` inteira.
- Se *ocorre* uma exceção durante a execução do bloco `try`, o Python volta e executa o bloco `finally`, mas então propaga a exceção para uma instrução `try` mais acima ou para a rotina de tratamento de nível superior padrão. O programa não retoma a execução após a instrução `try`.

A forma `try/finally` é útil quando você quer ter certeza absoluta de que uma ação vai acontecer após a execução de algum código, independente do comportamento de exceção do programa.

Note que a cláusula finally não pode ser usada na mesma instrução try que except e else; portanto, é melhor considerá-la como uma forma de instrução distinta.

Exemplo: desenvolvendo ações de término com try/finally

Vimos exemplos simples de try/finally anteriormente. Aqui está um exemplo mais realista que ilustra uma função típica dessa instrução:

```
MyError = "my error"

def stuff(file):
    raise MyError

file = open('data', 'r')          # Abre um arquivo já existente.
try:
    stuff(file)                   # Lança exceção
finally:
    file.close()                  # Sempre fecha o arquivo.
...                               # Continua aqui, se não houver exceção.
```

Nesse código, encerramos uma chamada para uma função de processamento de arquivo em um bloco try com uma cláusula finally, para garantir que o arquivo seja sempre fechado, lance a função uma exceção ou não.

A função desse exemplo em particular não tem tanta utilidade (ela apenas lança uma exceção), mas encerrar chamadas em instruções try/finally é uma boa maneira de garantir que suas atividades de finalização (isto é, término) sejam sempre executadas. O Python sempre executa o código presente em seus blocos finally, independente de uma exceção acontecer no bloco try ou não.* Por exemplo, se a função aqui não lançasse uma exceção, o programa ainda executaria o bloco finally para fechar seu arquivo e, então, continuaria após a instrução try inteira.

A INSTRUÇÃO raise

Para lançar exceções explicitamente, você escreve instruções raise. Sua forma geral é simples – a palavra raise, seguida opcionalmente do nome da exceção a ser lançada e um item de dados extra a ser passado com a exceção:

```
raise <nome>                    # Lança uma exceção manualmente.
raise <nome>, <dados>           # Também passa dados extra para o capturador.
raise                           # Lança novamente a exceção mais recente.
```

A segunda forma permite que você passe um item de dados extra junto com a exceção, para fornecer detalhes para a rotina de tratamento. Na instrução raise, os dados são listados após o nome da exceção. Na instrução try, os dados são obtidos pela inclusão de uma variável para recebê-los. Por exemplo, em except name,X:, X receberá o item de dados extra listado na instrução raise. A terceira forma de raise simplesmente lança novamente a exceção corrente. Isso é útil se você quer propagar uma exceção capturada para outra rotina de tratamento.

Então, o que é um nome de exceção? Pode ser o nome de uma exceção interna do escopo interno (por exemplo, IndexError) ou o nome de um objeto string arbitrário que você atribuiu em seu programa. Ele também pode referenciar uma classe definida pelo usuário ou uma

* A não ser que o Python falhe completamente, é claro. O Python faz um bom trabalho de evitar falhas, verificando todos os erros possíveis quando um programa é executado. Quando um programa tem uma falha séria, freqüentemente ela é devida a um erro no código de extensão C vinculado, fora do raio de ação do Python.

instância de classe – uma possibilidade que generaliza ainda mais os formatos da instrução `raise`. Vamos deixar os detalhes dessa generalização para depois que tivermos uma chance de estudar as exceções de classe, no próximo capítulo.

Independente de como você nomeia suas exceções, elas são sempre identificadas por objetos normais e, no máximo, uma está ativa em dado momento. Uma vez capturada por uma cláusula `except`, em qualquer lugar no programa, uma exceção desaparece (não se propaga para outra instrução `try`), a menos que seja lançada novamente por outra instrução `raise` ou por um erro.

Exemplo: lançando e capturando exceções definidas pelo usuário

Os programas em Python podem lançar exceções internas e definidas pelo usuário, com a instrução `raise`. Em sua forma mais simples, as exceções definidas pelo usuário são objetos string, como aquele em que a variável `MyBad` é atribuída no código a seguir:

```
MyBad = "oops"

def stuff():
    raise MyBad                     # Lança a exceção manualmente.

try:
    stuff()                         # Lança a exceção
except MyBad:
    print 'got it'                  # Trata da exceção aqui.
...                                 # Retoma a execução aqui.
```

Desta vez, a instrução `raise` ocorre dentro de uma função, mas isso não faz nenhuma diferença real – o controle pula para o bloco `except` imediatamente. Note que as exceções definidas pelo usuário são capturadas com instruções `try`, exatamente como as exceções internas.

Exemplo: passando dados extras com raise

Conforme sugerido anteriormente, as instruções `raise` podem passar um item de dados extra junto com a exceção para uso em uma rotina de tratamento. Em geral, os dados extras permitem que você envie informações contextuais sobre a exceção para uma rotina de tratamento. Se você estivesse escrevendo um analisador de arquivo de dados, por exemplo, poderia lançar uma exceção de erro de sintaxe no caso de erros e passar também um objeto que fornecesse informações sobre a linha e o arquivo para a rotina de tratamento (veremos um exemplo disso posteriormente nesta parte).

Rigorosamente falando, toda exceção tem os dados extras: de forma muito parecida com os valores de retorno de função, o padrão é o objeto especial `None`, caso nada seja passado explicitamente. O código a seguir, *raisedata.py*, ilustra esse conceito em ação:

```
MyException = 'Error'                  # Objeto string

def raiser1():
    raise myException, "hello"         # Lança, passa dados.

def raiser2():
    raise myException                  # Lança, None implícito.

def tryer(func):
    try:
        func()
    except myException, extraInfo:     # Executa func; captura exceção+dados.
        print 'got this', extraInfo
```

```
% python
>>> from raisedata import *
>>> tryer(raiser1)                  # Dados extras passados explicitamente
got this: hello
>>> tryer(raiser2)                  # O dado extra é None por padrão.
got this: None
```

Aqui, a função `tryer` sempre solicita o objeto dados extras; ele volta como uma string explícita de `raiser1`, mas tem `None` como padrão na instrução `raise` de `raiser2`. Posteriormente, veremos que o mesmo gancho pode ser usado para acessar instâncias lançadas em conjunto com exceções baseadas em classe.

Exemplo: propagando exceções com raise

Uma instrução `raise` sem um nome de exceção ou valor de dados extra simplemente lança a exceção corrente novamente. Ela é normalmente usada se você precisa capturar e tratar de uma exceção, mas não quer que ela desapareça em seu código:

```
>>> try:
...     raise IndexError, 'spam'
... except IndexError:
...     print 'propagating'
...     raise
...
propagating
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
IndexError: spam
```

Executando-se uma instrução `raise` dessa maneira, a exceção será lançada novamente e, assim, propagada para uma rotina de tratamento mais acima ou para a rotina de tratamento padrão no nível superior, a qual interrompe o programa com uma mensagem de erro padrão.

A INSTRUÇÃO assert

Como um caso um tanto especial, o Python inclui a instrução `assert`. Ela é principalmente um atalho sintático para um padrão de utilização comum de `raise` e pode ser considerada como uma instrução `raise` *condicional*. Uma instrução da forma:

```
assert <teste>, <dados>              # A parte <dados> é opcional.
```

funciona como o código a seguir:

```
if __debug__:
   if not <teste>:
      raise AssertionError, <dados>
```

Em outras palavras, se o teste é avaliado como falso, o Python lança uma exceção, com o item de dados como os dados extras da exceção (se forem fornecidos). Assim como todas as exceções, a exceção de erro de afirmação (`assertion error`) lançada eliminará seu programa, se não for capturada com uma instrução `try`.

Como um recurso adicional, as instruções `assert` também podem ser removidas do código de byte do programa compilado, caso seja usado o flag de linha de comando `-O` do Python, otimizando assim o programa. `AssertionError` é uma exceção interna e o flag `__debug__` é um nome interno configurado automaticamente como `1` (verdadeiro), a não ser que seja usado o flag `-O`.

25

Objetos Exceção

Até aqui, fomos deliberadamente vagos a respeito do que *é* realmente uma exceção. O Python generaliza a noção de exceções – elas podem ser identificadas por objetos string ou classe. Ambos têm méritos, mas as classes tendem a fornecer uma solução melhor quando se trata de manter hierarquias de exceção.

EXCEÇÕES BASEADAS EM STRINGS

Em todos os exemplos que vimos até este ponto, as exceções definidas pelo usuário eram strings. Essa é a maneira mais simples de escrever uma exceção – qualquer valor de string pode ser usado para identificar uma exceção:

```
>>> myexec = "My exception string"
>>> try:
...     raise myexec
... except myexec:
...     print 'caught'
...
caught
```

Tecnicamente, a exceção é identificada pelo *objeto* string e não pelo valor da string – você deve usar a mesma variável (isto é, referência) para lançar e capturar a exceção (vamos expandir essa idéia em um problema na conclusão da Parte VII). Aqui, o nome da exceção `myexec` é apenas uma variável normal – ela pode ser importada de um módulo etc. O texto da string é quase irrelevante, exceto que ele aparece nas mensagens de erro padrão:

```
>>> raise myexec
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
My exception string
```

O texto da exceção de string aqui é impresso como a mensagem de exceção. Se suas exceções de string podem imprimir dessa forma, você desejará usar texto mais significativo do que a maioria dos exemplos mostrados neste livro.

EXCEÇÕES BASEADAS EM CLASSE

As strings são uma maneira simples de definir suas exceções. As exceções também podem ser identificadas com classes. Assim como alguns outros assuntos que conhecemos neste livro, as exceções de classe são um tópico avançado que você pode usar ou não no Python 2.2. Entretanto, as classes têm algum valor agregado que merece ser visto rapidamente. Em particular, elas nos permitem identificar *categorias* de exceção que são mais flexíveis para usar e manter do que as strings simples. Além disso, as classes provavelmente se tornarão a maneira prescrita de identificar suas exceções no futuro.

A principal diferença entre exceções de string e de classe está relacionada com a maneira pela qual as exceções lançadas correspondem às cláusulas `except` em instruções `try`:

- As exceções de string correspondem à *identidade do objeto* simples: a exceção lançada corresponde às cláusulas `except` pelo teste `is` do Python (e não `==`).
- As exceções de classe correspondem aos *relacionamentos da superclasse*: a exceção lançada corresponde a uma cláusula `except`, se essa cláusula `except` nomeia a classe da exceção ou qualquer superclasse dela.

Isto é, quando a cláusula `except` de uma instrução `try` lista uma superclasse, ela captura instâncias dessa superclasse, assim como instâncias de todas as suas subclasses mais abaixo na árvore de classes. O resultado é que as exceções de classe suportam a construção de hierarquias de exceção: as superclasses tornam-se nomes de *categoria* e as subclasses tornam-se tipos específicos de exceções dentro de uma categoria. Nomeando uma superclasse de exceção geral, uma cláusula `except` pode capturar uma categoria inteira de exceções – qualquer subclasse mais específica corresponderá.

Exemplo de exceção de classe

Vamos ver um exemplo para saber como as exceções de classe funcionam no código. No arquivo a seguir, *classexc.py*, definimos uma superclasse `General` e duas subclasses dela, chamadas `Specific1` e `Specific2`. Estamos ilustrando a noção de categorias de exceção aqui: `General` é um nome de categoria e suas duas subclasses são tipos específicos de exceções dentro da categoria. As rotinas de tratamento que capturam `General` também capturarão todas as subclasses dela, incluindo `Specific1` e `Specific2`.

```
class General:            pass
class Specific1(General):  pass
class Specific2(General):  pass

def raiser0():
    X = General()        # Lança instância de superclasse.
    raise X

def raiser1():
    X = Specific1()      # Lança instância de subclasse.
    raise X

def raiser2():
    X = Specific2()      # Lança instância de subclasse diferente.
    raise X

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General:      # Corresponde a General ou a qualquer subclasse dela.
```

```
        import sys
        print 'caught:', sys.exc_type

C:\python> python classexc.py
caught: __main__.General
caught: __main__.Specific1
caught: __main__.Specific2
```

Note que, aqui, chamamos classes para criar *instâncias* nas instruções `raise`. Conforme veremos quando formalizarmos as formas da instrução `raise`, posteriormente nesta seção, uma instância está sempre presente ao se lançar exceções baseadas em classe. Esse código também inclui funções que lançam instâncias de todas as três classes como exceções e uma instrução `try` de nível superior que chama as funções e captura exceções de `General`. A mesma instrução `try` captura `General` e as duas exceções específicas, pois as duas exceções específicas são subclasses de `General`.

Por que exceções de classe?

Como existem apenas três exceções possíveis no exemplo da seção anterior, ele não faz justiça à utilidade das exceções de classe. Na verdade, podemos obter os mesmos efeitos desenvolvendo uma lista de nomes de exceção de string entre parênteses dentro da cláusula `except`. O arquivo *stringexc.py* mostra como:

```
General =   'general'
Specific1 =  'specific1'
Specific2' = 'specific2'

def raiser0(): raise General
def raiser1(): raise Specific1
def raiser2(): raise Specific2

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except (General, Specific1, Specific2): # Captura qualquer uma delas.
        import sys
        print 'caught:' sys.exc_type

C:\python> python stringexc.py
caught: general
caught: specific1
caught: specific2
```

Mas para hierarquias de exceção maiores ou mais altas, capturar categorias usando classes pode ser mais fácil do que listar cada membro de uma categoria em uma única cláusula `except`. Além disso, as hierarquias de exceção podem ser estendidas adicionando-se novas subclasses, sem prejudicar o código existente.

Suponha que você desenvolva uma biblioteca de programação numérica em Python, para ser usada por um grande número de pessoas. Enquanto está escrevendo sua biblioteca, você identifica duas coisas que podem dar errado com números em seu código – divisão por zero e estouro numérico (overflow). Você documenta isso como as duas exceções que sua biblioteca pode lançar e as define como strings simples em seu código:

```
divzero =   'Division by zero error in library'
oflow =     'Numeric overflow error in library'
...
raise divzero
```

Agora, quando as pessoas usarem sua biblioteca, normalmente elas encerrarão chamadas para suas funções ou classes em instruções `try` que capturam suas duas exceções (se elas não capturarem suas exceções, as exceções da biblioteca eliminarão o código delas):

```
import mathlib
...
try:
    mathlib.func(...)
except (mathlib.divzero, mathlib.oflow):
    ...relata e recupera...
```

Isso funciona bem e as pessoas usam sua biblioteca. Seis meses depois, você revisa sua biblioteca. No processo, você identifica outra coisa que pode dar errado – underflow – e adiciona isso como uma nova exceção de string:

```
divzero = 'Division by zero error in library'
oflow   = 'Numeric overflow error in library'
uflow   = 'Numeric underflow error in library'
```

Infelizmente, quando relança seu código, você acabou de criar um problema de manutenção para seus usuários. Supondo que eles listem suas exceções explicitamente, agora precisam voltar e alterar cada lugar onde chamam sua biblioteca, para incluir o nome de exceção recentemente adicionado:

```
try:
    mathlib.func(...)
except (mathlib.divzero, mathlib.oflow, mathlib.uflow):
    ...relata e recupera...
```

Agora, talvez isso não seja o fim do mundo. Se sua biblioteca é usada apenas internamente, você mesmo pode fazer as alterações. Você também poderia distribuir um script em Python que tentasse corrigir o código automaticamente (ele teria poucas dezenas de linhas e acertaria pelo menos em parte do tempo). Contudo, se muitas pessoas tiverem que alterar seu código, sempre que você alterar seu conjunto de exceções, essa não será exatamente a mais cortês das políticas de atualização.

Seus usuários poderiam tentar evitar essa armadilha escrevendo cláusulas `except` vazias:

```
try:
    mathlib.func(...)
except:                               # Captura tudo aqui.
    ...relata e recupera...
```

O problema desse artifício é que ele pode capturar mais do que o desejado – até coisas como erros de memória e saídas do sistema lançam exceções, e você quer que essas coisas passem e não sejam capturadas e classificadas erroneamente como um erro de biblioteca. Como regra geral, normalmente é melhor ser específico do que geral nas rotinas de tratamento de exceção (uma idéia que reveremos nos problemas).

Então, o que fazer? As exceções de classe resolvem esse dilema completamente. Em vez de definir as exceções de sua biblioteca como um conjunto simples de strings, organize-as em uma árvore de classes, com uma superclasse comum para conter a categoria inteira:

```
class NumErr: pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass
...
raise Divzero()
```

Desse modo, os usuários de sua biblioteca só precisam listar a superclasse (isto é, a categoria) comum para capturar todas as exceções – tanto agora como no futuro:

```
import mathlib
...
try:
    mathlib.func(...)
except mathlib.NumErr:
    ...relata e recupera...
```

Quando você volta e mexe em seu código novamente, as novas exceções são adicionadas como novas subclasses da superclasse comum:

```
class Uflow(NumErr): pass
```

O resultado final é que o código de usuário que captura as exceções de sua biblioteca continuará funcionando *sem alteração*. Na verdade, você fica então livre para adicionar, excluir e alterar suas exceções arbitrariamente no futuro – contanto que os clientes nomeiem a superclasse, eles ficam isolados das alterações em seu conjunto de exceções. Em outras palavras, as exceções de classe oferecem uma resposta melhor do que as strings para os problemas de manutenção. As exceções baseadas em classe também suportam retenção de estado e herança de maneiras que as strings não conseguem – um conceito que exploraremos por meio de exemplo, posteriormente nesta seção.

Classes de exceção internas

Os exemplos da seção anterior não mostraram tudo. Embora as exceções definidas pelo usuário possam ser identificadas por objetos string ou classe, todas as exceções internas que o Python em si pode lançar são objetos classe predefinidos, em vez de strings. Além disso, elas são organizadas em uma hierarquia pouco profunda, com categorias de superclasse gerais e tipos de subclasse específicos, muito parecida com a árvore de classes de exceção da seção anterior.

Todas as exceções familiares que você já viu (por exemplo, `SyntaxError`) são, na verdade, apenas classes predefinidas, disponíveis como nomes internos (no módulo `__builtin__`) e como atributos do módulo `exceptions` da biblioteca padrão. Além disso, o Python organiza as exceções internas em uma hierarquia para suportar uma variedade de modos de captura. Por exemplo:

`Exception`
: Superclasse-raiz de nível superior de exceções

`StandardError`
: A superclasse de todas as exceções de erro internas

`ArithmeticError`
: A superclasse de todos os erros numéricos

`OverflowError`
: Uma subclasse que identifica um erro numérico específico

E assim por diante – você pode ler mais sobre essa estrutura no manual de biblioteca ou no texto de ajuda do módulo `exceptions` (consulte o Capítulo 11 para informações sobre `help`):

```
>>> import exceptions
>>> help(exceptions)
...muito texto omitido...
```

A árvore de classes interna permite que você escolha o quanto suas rotinas de tratamento serão específicas ou gerais. Por exemplo, a exceção interna `ArithmeticError` é uma superclasse para exceções mais específicas, como `OverflowError` e `ZeroDivisionError`. Listando `Arithmeti-`

`cError` em uma instrução `try`, você capturará qualquer tipo de erro numérico gerado; listando apenas `OverflowError`, você interceptará apenas esse tipo de erro específico e nenhum outro.

Analogamente, como `StandardError` é a superclasse de todas as exceções de erro internas, geralmente você pode usá-la para selecionar entre erros internos e exceções definidas pelo usuário em uma instrução `try`:

```
try:
    action()
except StandardError:
    ... trata de erros do Python...
except:
    ...trata de exceções de usuário...
else:
    ...trata do caso de nenhuma exceção...
```

Você também pode quase simular uma cláusula `except` vazia (que captura tudo), capturando a classe-raiz `Exception`, mas não completamente – atualmente, as exceções de string, assim como as exceções definidas pelo usuário independentes, não são subclasses da classe-raiz `Exception`.* Use você ou não as categorias da árvore de classes interna, ela serve como um bom exemplo. Usando técnicas semelhantes para exceções de classe em seu próprio código, você pode fornecer conjuntos de exceção flexíveis e facilmente modificados.

Fora isso, as exceções internas geralmente são indistinguíveis das strings. Na verdade, normalmente você não precisa se preocupar com o fato de que elas são classes, a não ser que suponha que as exceções internas são strings e tente concatenar sem converter (por exemplo, `KeyError+"spam"` falha, mas `str(KeyError)+"spam"` funciona).

Especificando texto de exceção

Quando conhecemos as exceções baseadas em string, no início desta seção, vimos que o texto da string aparece na mensagem de erro padrão, quando a exceção não é capturada. Para uma exceção de classe não capturada, por padrão, você recebe o nome da classe e uma tela não muito bonita do objeto instância lançado:

```
>>> class MyBad: pass

>>> raise MyBad()
TraceBack (most recent call last):
  File "<pyshell#30>", line 1, in ?
    raise MyBad
MyBad: <__main__.MyBad instance at 0x00B58980>
```

Para fazer melhor, defina o método de sobrecarga de representação de string `__repr__` ou `__str__` em sua classe, para retornar a string que você deseja exibir para sua exceção, caso ela atinja a rotina de tratamento padrão no nível superior:

```
>>> class MyBad:
...     def __repr__(self):
...         return "Sorry--my mistake!"
...
>>> raise MyBad()
```

* Note que a documentação atual do Python diz que "Recomenda-se que as exceções baseadas em classes definidas pelo usuário sejam derivadas da classe `Exception`, embora atualmente isso não seja obrigatório". Isto é, as subclasses de `Exception` são preferidas para classes de exceção independentes, mas não exigidas. O programador defensivo poderia deduzir que pode ser uma boa idéia adotar essa política de qualquer forma.

```
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in ?
    raise MyBad()
MyBad: Sorry--my mistake!
```

O método de sobrecarga `__repr__` é chamado para imprimir, e pelos pedidos de conversão de string feitos para instâncias de sua classe. Consulte a seção "Sobrecarga de operador", no Capítulo 21.

Enviando dados extras em instâncias

Além de suportar hierarquias flexíveis, as exceções de classe também fornecem armazenamento para informações *de estado* extras como atributos de instância. Quando uma exceção baseada em classe é lançada, o Python passa automaticamente o objeto instância de classe, junto com a exceção, como o item de dados extra. Assim como acontece com as exceções de string, você pode acessar a instância lançada listando uma variável extra na instrução `try`. Isso fornece um gancho natural para prover dados e comportamento para a rotina de tratamento.

Exemplo: dados extras com classes e strings

Vamos demonstrar a noção de dados extras por meio de um exemplo e, no processo, comparar as estratégias baseadas em classe e em string. Um programa que analisa arquivos de dados poderia sinalizar um erro de formatação lançando uma instância de exceção preenchida com detalhes extras sobre o erro:

```
>>>class FormatError:
...      def __init__(self, line, file):
...          self.line = line
...          self.file = file
...
>>> def parser():
...     # quando erro é encontrado
...     raise FormatError(42, file='spam'.txt')
...
>>> try:
...     parser()
... except FormatError, X:
...      print 'Error at', X.file, X.line
...
Error at spam.txt 42
```

Na cláusula `except` aqui, a variável `X` recebe uma referência para a instância gerada onde a exceção foi lançada. Na prática, contudo, isso não é significativamente mais conveniente do que passar objetos compostos (por exemplo, tuplas, listas ou dicionários) como dados extras com exceções de string e, por si só, pode não ser atraente o suficiente para garantir exceções baseadas em classe:

```
>>> formatError = 'formatError'

>>> def parser():
...     # quando erro é encontrado
...     raise formatError {'line':42, 'file':'spam.txt'}
...
>>> try:
...     parser()
```

```
... except formatError, X:
...     print 'Error at', X['file'], X['line']
...
Error at spam.txt 42
```

Desta vez, a variável `X` na cláusula `except` recebe o dicionário de detalhes extras listados na instrução `raise`. O resultado é semelhante, sem a necessidade de se desenvolver uma classe no processo. A estratégia da classe poderia ser mais conveniente, caso a exceção também deva ter comportamento – a classe de exceção também pode definir *métodos* para serem chamados na rotina de tratamento:

```
class FormatError:
     def __init__(self, line, file):
         self.line = line
         self.file = file
     def logerror(self):
         log = open('formaterror.txt', 'a')
         print >> log, 'Error at', self.file, self.line

def parser():
         raise FormatError, exc:

try
    parser
except FormatError, exc:
    exc.logerror()
```

Em tal classe, métodos (como `logerror`) também podem ser *herdados* de superclasses e os atributos de instância (como `line` e `file`) fornecem um lugar para salvar o *estado* para uso em chamadas de método posteriores. Aqui, podemos imitar grande parte desse efeito passando *funções* simples na estratégia baseada em string:

```
formatError = "formatError"

def logerror(line, file):
    log = open('formaterror.txt', 'a')
    print >> log, 'Error at', file, line

def parser():
    raise formatError, (41, 'spam.txt', logerror)

try:
    parser()
except formatError, data:
    data[2](data[0], data[1])          # Ou simplesmente: logerror()
```

Naturalmente, tais funções não participariam na herança, como acontece com os métodos de classe, e não poderiam manter o estado em atributos de instância (`lambdas` e variáveis globais normalmente são o melhor que podemos fazer para funções sem estado). É claro que poderíamos passar uma instância de classe nos dados extras de exceções baseadas em string, para obter o mesmo efeito. Mas se fomos até aqui para imitar as exceções baseadas em classe, também poderíamos adotá-las – de qualquer forma estaríamos desenvolvendo uma classe.

Em geral, a escolha entre exceções baseadas em string e baseadas em classe é muito parecida com a opção de usar classes; nem todo programa exige o poder da POO. As exceções baseadas em string representam uma ferramenta mais simples para tarefas mais simples. As exceções baseadas em classe tornam-se mais úteis para definir *categorias* e em aplicativos avançados que podem tirar proveito da retenção do *estado* e da *herança* de atributos. Como

26

Projetando com Exceções

Este capítulo finaliza a Parte VII com uma coleção de tópicos e exemplos de projeto com exceção, seguida dos problemas e exercícios desta parte. Como este capítulo também encerra o material básico da linguagem deste livro, ele inclui ainda um breve panorama das ferramentas de desenvolvimento, como forma de migração para o restante do livro.

ANINHANDO ROTINAS DE TRATAMENTO DE EXCEÇÃO

Nossos exemplos até aqui usaram apenas uma instrução `try` para capturar exceções, mas o que acontece se uma instrução `try` está fisicamente aninhada dentro de outra? Quanto a isso, o que acontece se uma instrução `try` chama uma função que executa outra instrução `try`? Tecnicamente, as instruções `try` podem ser *aninhadas*, tanto em termos de sintaxe quanto no fluxo de controle em tempo de execução em seu código.

Esses dois casos podem ser entendidos se você perceber que o Python *empilha* as instruções `try` em tempo de execução. Quando uma exceção é lançada, o Python retorna para a instrução `try` com uma cláusula `except` correspondente, em que entrou mais recentemente. Como cada instrução `try` deixa um marcador, o Python pode pular para instruções `try` anteriores inspecionando os marcadores empilhados. Esse aninhamento de rotinas de tratamento ativas é o que queremos dizer com rotinas de tratamento "superiores" – as instruções `try` em que se entrou anteriormente no fluxo de execução do programa.

Por exemplo, a Figura 26-1 ilustra o que ocorre quando instruções `try/except` são aninhadas em tempo de execução. Como a quantidade de código que pode estar em um bloco de cláusula `try` pode ser grande (por exemplo, chamadas de função), normalmente ele executará outro código que pode estar observando a mesma exceção. Quando uma exceção é finalmente lançada, o Python pula para a instrução `try` que nomeia essa exceção em que entrou mais recentemente, executa as cláusulas `except` dessa instrução e, então, retoma após essa instrução `try`.

Quando uma exceção é capturada, sua vida termina – o controle não volta para todas as instruções `try` correspondentes que nomeiam a exceção, mas apenas para *uma*. Na Figura 26-1, por exemplo, a instrução `raise` na função `func2` envia o controle para a rotina de tratamento que está em `func1` e, então, o programa continua dentro de `func1`.

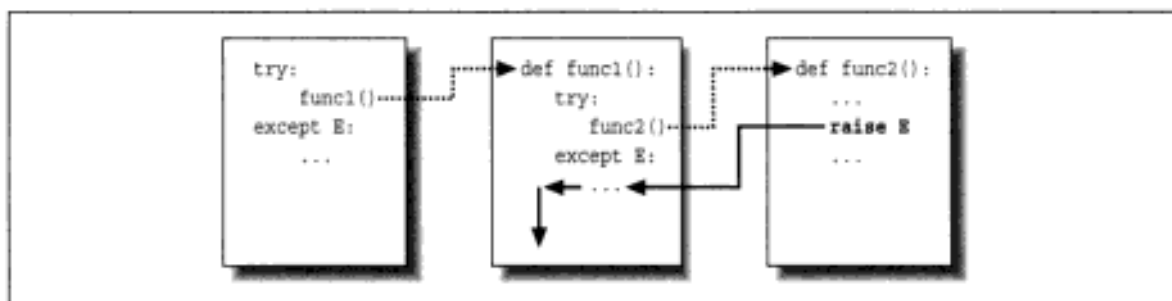



Figura 26-1. Instruções `try/except` aninhadas.

Em contraste, quando são usadas instruções `try/finally`, o controle executa o bloco `finally` na ocorrência de exceções, mas então continua a propagar a exceção para outras instruções `try` ou para a rotina de tratamento padrão de nível superior (a impressora de mensagem de erro padrão). Conforme ilustra a Figura 26-2, as cláusulas `finally` não eliminam a exceção – elas apenas especificam código a ser executado ao sair, durante o processo de propagação de exceção. Se houver muitas cláusulas `try/finally` ativas no momento em que ocorrer uma exceção, *todas* elas serão executadas (a menos que um bloco `try/except` capture a exceção em algum lugar ao longo do caminho).

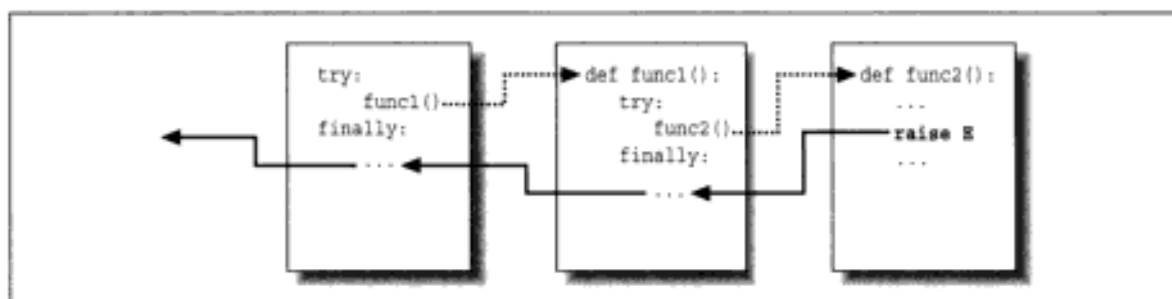



Figura 26-2 Instruções `try/finally` aninhadas.

Exemplo: aninhamento de fluxo de controle

Vamos ver um exemplo para tornar esse conceito de aninhamento mais concreto. O módulo a seguir, no arquivo *nestexc.py*, define duas funções: `action2` é desenvolvida para lançar uma exceção (você não pode somar números e seqüências) e `action1` encerra uma chamada para `action2` em uma rotina de tratamento `try`, para capturar a exceção:

```
def action2 ():
    print 1 + []               # Gera TypeError.

def action1():
    try:
        action2()
except TypeError:              # Instrução try correspondente mais recente
        print 'inner try'

try:
    action1()
except TypeError:              # Aqui, somente se action1 lançar novamente.
    print 'outer try'

% python nestexc.py
inner try
```

Note, contudo, que o código do módulo de nível superior na parte final do arquivo também encerra uma chamada para `action1` em uma rotina de tratamento `try`. Quando `action2` lançar a exceção TypeError, haverá duas instruções `try` ativas – a que está em `action1` e a que está no nível superior do módulo. O Python escolhe e executa apenas a mais recente com uma cláusula `except` correspondente, a qual, neste caso, é a instrução `try` que está dentro de `action1`.

Em geral, o lugar para onde uma exceção acaba pulando depende do fluxo de controle pelo programa em tempo de execução. Em outras palavras, para saber para onde irá, você precisa saber *onde esteve* – as exceções são mais uma função do fluxo de controle do que uma sintaxe de instrução.

Exemplo: aninhamento sintático

Também é possível aninhar instruções `try` sintaticamente:

```
try:
    try:
        action2()
    except TypeError:              #Instrução try correspondente mais recente
        print 'inner try'
except TypeError:                  # Aqui, somente se a rotina de tratamento
                                   # aninhada lançar novamente.
    print 'outer try'
```

Mas, na realidade, isso apenas configura a mesma estrutura de aninhamento de rotina de tratamento e se comporta de forma idêntica ao exemplo anterior. Na verdade, o aninhamento sintático funciona exatamente como os casos que esboçamos nas figuras 26-1 e 26-2. A única diferença é que as rotinas de tratamento aninhadas são fisicamente incorporadas em um bloco `try` e não escritas em uma função chamada em outro lugar. Por exemplo, todas as rotinas de tratamento `finally` aninhadas são executadas no caso de uma exceção, sejam aninhadas sintaticamente ou partes fisicamente separadas pelo fluxo de tempo de execução de seu código:

```
>>> try:
...       try:
...           raise IndexError
...       finally:
...          print 'spam'
... finally:
...       print 'SPAM'
...
spam
SPAM
Traceback (most recent call last):
  File "<stdin>", line3, in ?
IndexError
```

Veja na Figura 26-2 uma ilustração gráfica do funcionamento desse código. É o mesmo efeito, mas aqui a lógica da função foi colocada em linha como instruções aninhadas. Para um exemplo mais útil de aninhamento sintático em funcionamento, considere o arquivo a seguir, *except-finally.py:*

```
def raise1(): raise IndexError
def noraise(): return
def raise2(): raise SyntaxError

for func in (raise1, noraise, raise2):
    print '\n', func
```

```
try:
    try:
        func()
    except IndexError:
        print 'caught IndexError'
finally:
    print 'finally run'
```

Esse código captura uma exceção, se for lançada, e executa uma ação `finally` no momento do término, independente de uma exceção ter ocorrido ou não. Isso demora um pouco para digerir, mas o efeito é muito parecido com a combinação de cláusulas `except` e `finally` em uma única instrução `try`, mesmo que essa combinação seja sintaticamente inválida (elas são mutuamente exclusivas):

```
% python except-finally.py

<function raise1 at 0x00867DF8>
caught IndexError
finally run

<function noraise at 0x00868EB8>
finally run

<function raise2 at 0x00875B80>
finally run
Traceback (most recent call last):
  File "except-finally.py", line 9, in ?
    func()
  File "except-finally.py", line 3, in raise2
    def raise2(): raise SyntaxError
SintaxError
```

IDIOMAS DE EXCEÇÃO

Vimos os mecanismos existentes por trás das exceções. Agora, vamos ver algumas das outras maneiras como elas são normalmente usadas.

As exceções nem sempre são erros

No Python, todos os erros são exceções, mas nem todas as exceções são erros. Por exemplo, vimos no Capítulo 7 que os métodos de leitura de objeto arquivo retornam strings vazias no final de um arquivo. A função interna `raw`, que conhecemos pela primeira vez no Capítulo 3 e foi usada em um loop interativo no Capítulo 10, lê uma linha de texto do fluxo de entrada padrão (`sys.stdin`). Ao contrário dos métodos de arquivo, `raw_input` lança a exceção interna `EOFError` no final de um arquivo, em vez de retornar uma string vazia (uma string vazia de `raw_input` significa uma linha vazia).

Apesar de seu nome, a exceção `EOFError` (erro de fim de arquivo) é apenas um sinal nesse contexto e não um erro. Por causa desse comportamento, a não ser que o final de um arquivo deva terminar um script, `raw_input` freqüentemente aparece encerrada em uma rotina de tratamento `try` e aninhada em um loop, como no código a seguir.

```
while 1:
   try:
       line = raw_input()          # Lê linha de stdin.
   except EOFError:
```

Depurando com instruções try externas

Você também pode usar rotinas de tratamento de exceção para substituir o comportamento de tratamento de exceção de nível superior padrão do Python. Encerrando um programa inteiro (ou uma chamada para ele) em uma instrução `try` externa em seu código de nível superior, você pode capturar qualquer exceção que possa ocorrer enquanto seu programa é executado, subvertendo assim o término de programa padrão.

No código a seguir, a cláusula `except` vazia captura qualquer exceção não capturada lançada enquanto o programa executa. Para capturar a exceção real ocorrida, busque os atributos `exc_type` e `exc_value` do módulo interno `sys`; eles são configurados automaticamente com o nome e os dados extras da exceção corrente:*

```
try:
    ...executa o programa...
except:                        # Todas as exceções não capturadas ficam aqui.
    import sys
    print 'uncaught!', sys.exc_type, sys.exc_value
```

Essa estrutura é normalmente usada durante o desenvolvimento, para manter seu programa ativo mesmo após a ocorrência de erros – você pode executar testes adicionais sem a necessidade de reiniciar. Ela também é usada ao se testar código, conforme descrito na próxima seção.

Executando testes no processo

Você poderia combinar alguns desses padrões de desenvolvimento em um aplicativo de driver de teste que testa outro código dentro do mesmo processo:

```
import sys
log = open('testlog', 'a')
from testapi import moreTests, runNextTest, testName

def testdriver():
    while moreTests():
        try:
            runNextTest()
        except:
            print >> log, 'FAILED', testName(), sys.exc_type
        else:
            print >> log, 'PASSED', testName()
testdriver()
```

Aqui, a função `testdriver` circula por uma série de chamadas de teste (o módulo `testapi` foi deixado abstrato nesse exemplo). Como uma exceção não capturada em um caso de teste normalmente eliminaria esse driver de teste, precisamos encerrar as chamadas de caso de teste em uma instrução `try`, se quisermos continuar o processo de teste depois que um teste falhar. Como sempre, a cláusula `except` vazia captura qualquer exceção não capturada gerada por um caso de teste e usa `sys.exc_type` para registrar a exceção em um arquivo; a cláusula `else` é executada quando não ocorre nenhuma exceção – no caso do sucesso do teste.

* O módulo interno `traceback` permite que a exceção corrente seja processada de maneira genérica e a função `sys.exc_info()` retorna uma tupla contendo o tipo, os dados e o rastreamento da exceção corrente. `sys.exc_type` e `sys.exc_value` ainda funcionam, mas gerenciam uma única exceção global. `exc_info()` monitora as informações de exceção de cada segmento e, assim, é específica do segmento. Essa distinção só importa ao se usar múltiplos segmentos nos programas em Python (um assunto que está fora dos objetivos deste rodapé). Consulte o manual da biblioteca Python para ver mais detalhes.

Por que isto é relevante: verificações de erro

Uma maneira de ver por que as exceções são úteis é comparando os estilos de desenvolvimento em Python e em linguagens que não possuem exceções. Por exemplo, se você quiser escrever programas robustos na linguagem C, geralmente precisará testar valores de retorno ou códigos de status após cada operação que poderia se desviar:

```
doStuff()
{                                    # Programa em C:
     if (doFirstThing() == ERROR)    # Detecta erros por toda parte
          returnERROR;               # mesmo que não sejam tratados aqui.
     if (doNextThing() == ERROR)
          return ERROR;
     ...
     return doLastThing();
}

main()
{
     if (doStuff() == ERROR)
          badEnding();
     else
          goodEnding();
}
```

Na verdade, programas em C realistas freqüentemente têm tanto código dedicado à detecção de erros quanto para fazer o trabalho real. Mas no Python você não precisa ser tão metódico. Em vez disso, você pode encerrar trechos arbitrariamente grandes de um programa em rotinas de tratamento de exceção, e escrever as partes que realizam o trabalho real de forma a presumir que tudo está bem:

```
def doStuff()                 # Código Python
    doFirstThing()            # Não nos preocupamos com exceções aqui,
    doNextThing()             # de modo que não precisamos detectá-las aqui.
    ...
    doLastThing()

if __name__ == '__main__':
     try:
         doStuff()            # É aqui que nos preocupamos com os resultados;
    except:                   # portanto, é o único lugar em que devemos verificar.
         badEnding()
    else:
         goodEnding()
```

Como o controle pula imediatamente para uma rotina de tratamento, quando ocorre uma exceção, não há necessidade de aparelhar todo seu código para prevenir-se contra erros. Além disso, como o Python detecta erros automaticamente, seu código normalmente nem mesmo precisa verificar erros. A conclusão é que, de modo geral, as exceções permitem a você ignorar os casos incomuns e evitar muito código de verificação de erro.

```
def func():
    try:
        ...                          # IndexError é lançada aqui.
    except:
        ...                          # Mas tudo chega aqui e morre!
```

```
try:
    func()
except IndexError:                    # Necessário aqui
    ...
```

Talvez pior, esse código também poderia capturar exceções do sistema. Mesmo coisas como erros de memória, erros de programação, paradas de iteração e saídas de sistema, lançam exceções no Python. Tais exceções normalmente não devem ser interceptadas.

Por exemplo, normalmente, os scripts saem quando o controle ultrapassa o final do arquivo de nível superior. Entretanto, o Python também fornece a chamada interna `sys.exit` para permitir términos antecipados. Na verdade, isso funciona lançando a exceção interna `SystemExit` para finalizar o programa, de modo que as rotinas de tratamento `try/finally` são executadas ao sair e tipos especiais de programas podem interceptar o evento.* Por isso, uma instrução `try` com uma cláusula `except` vazia poderia impedir, sem saber, uma saída crucial, como no arquivo *exiter.py*:

```
import sys

def bye():
    sys.exit(40)                      # Erro crucial: cancela agora!

try:
    bye()
except:
    print 'got it'                    # Opa--ignoramos a saída
print 'continuing...'
```

```
% python exiter.py
got it
continuing...
```

Você simplesmente poderia não esperar todos os tipos de exceções que poderiam ocorrer durante uma operação. Na verdade, uma cláusula `except` vazia também capturaria erros de programação genuínos, os quais devem passar na maioria das vezes:

```
mydictionary = {...}
...
try:
    x = myditctionary['spam']         # Opa: grafado de forma errada
except:
    x= None                           # Presume que recebemos KeyError.
...continua aqui...
```

Aqui, o codificador presume que o único tipo de erro que pode ocorrer ao indexar um dicionário é um erro de chave. Mas, como o nome `myditctionary` foi grafado de forma errada (deveria ser `mydictionary`), o Python lança uma exceção `NameError` (em vez da referência de nome indefinido) que será silenciosamente capturada e ignorada pela rotina de tratamento. O evento preencherá incorretamente um padrão para o acesso do dicionário, mascarando o erro do programa. Se isso acontece em um código que está longe do lugar onde os valores buscados são usados, pode se transformar em uma tarefa de depuração muito interessante.

* Uma chamada relacionada, `os._exit`, também termina um programa, mas é um término imediato – ela pula ações de limpeza e não pode ser interceptada com `try/except` ou `try/finally`. Normalmente, ela é usada apenas em processos filhos gerados – um assunto fora dos objetivos deste livro. Consulte o manual da biblioteca ou o livro *Programming Python*, Second Edition (O'Reilly) para ver os detalhes.

Como regra geral, seja específico em suas rotinas de tratamento – as cláusulas `except` vazias são úteis, mas potencialmente propensas a erros. No último exemplo, normalmente você deve escrever `except KeyError`: para tornar suas intenções explícitas e evitar a interceptação de eventos não relacionados, por exemplo. Em scripts mais simples, o potencial para problemas pode não ser significativo o bastante para superar a conveniência de uma função que captura tudo. Mas, em geral, as rotinas de tratamento geralmente são problemáticas.

Capturando coisas de menos

Inversamente, as rotinas de tratamento também não devem ser específicas demais. Ao listar exceções específicas em uma instrução `try`, você captura apenas o que realmente lista. Isso também não é necessariamente algo ruim, mas se um sistema evoluir para lançar outras exceções no futuro, talvez você precise voltar e adicioná-las nas listas de exceção por toda parte no código.

Por exemplo, a rotina de tratamento a seguir foi escrita para tratar de `myerror1` e `myerror2` como casos normais, e tratar de tudo mais como erro. Se uma exceção `myerror3` for adicionada no futuro, ela será processada como um erro, a não ser que você atualize a lista de exceções:

```
try:
    ...
except (myerror1, myerror2):      # E se eu adicionar myerror3?
    ...                           # Não são erros
else:
    ...                           # Presumido como sendo um erro
```

O uso cuidadoso de exceções baseadas em classe pode fazer essa armadilha desaparecer completamente. Conforme aprendemos no capítulo anterior, se você captura uma superclasse geral, pode adicionar e lançar subclasses mais específicas no futuro, sem ter de estender as listas de cláusula `except` manualmente:

```
try:
    ...
except SuccessCategoryName:           # E se eu adicionar myerror3?
    ...                               # Não são erros
else:
    ...                               # Presumido como sendo um erro
```

Use você classes aqui ou não, um pouco de projeto faz muita diferença. A moral da história é que você precisa ter o cuidado de não ser geral ou específico demais nas rotinas de tratamento de exceção e escolher a granularidade do encerramento de sua instrução `try` sabiamente. Especialmente em sistemas maiores, as políticas de exceção devem fazer parte do projeto global.

PROBLEMAS DAS EXCEÇÕES

Não há muitas possibilidades de erro com as exceções, mas aqui estão duas indicações gerais sobre uso, uma das quais resume conceitos que já conhecemos.

As exceções de string correspondem por identidade e não por valor

Quando uma exceção é lançada (por você ou pelo próprio Python), o Python procura a instrução `try` com uma cláusula `except` correspondente em que entrou mais recentemente, onde correspondente significa o mesmo objeto string, a mesma classe ou uma superclasse da classe lançada. É importante notar que a correspondência é realizada pela identidade e não pela igualdade. Por exemplo, suponha que definamos dois objetos string que queremos lançar como exceções:

```
>>> ex1 = 'Error: Spam Exception'
>>> ex2 = 'Error: Spam Exception'
>>>
>>> ex1 == ex2, ex1 is ex2
(1, 0)
```

Aplicar o teste `==` retorna verdadeiro (1), pois eles têm valores iguais, mas `is` retorna falso `(0)`, pois são dois objetos string distintos na memória. Agora, uma cláusula `except` que nomeia o mesmo objeto string sempre corresponderá:

```
>>> try:
...     raise ex1
... except ex1:
...     print 'got it'
...
got it
```

Mas uma cláusula que liste um valor igual, mas não um objeto idêntico, falhará (supondo que os valores de string sejam longos o suficiente para anular o mecanismo de captura de objeto string do Python, que foi descrito nos capítulos 4 e 7:

```
>>> try:
...     raise ex1
... except ex2:
...     print 'Got it'
...
Traceback (innermost last):
  File "<stdin>", line 2, in ?
    raise ex1
Error: Spam Exception
```

Aqui, a exceção não é capturada; portanto, o Python sobe para o nível superior do processo e imprime um rastreamento de pilha e o texto da exceção, automaticamente. Para strings, certifique-se de usar o mesmo objeto nas instruções `raise` e `try`. Para exceções de classe, o comportamento é semelhante, mas o Python generaliza a noção de correspondência de exceção para incluir relacionamentos de superclasse.

Capturando a coisa errada

Talvez os problemas mais comuns relacionados às exceções envolvam as diretrizes de projeto da seção anterior. Lembre-se de tentar evitar as cláusulas `except` vazias (ou você poderá capturar coisas como saídas de sistema) e cláusulas `except` demasiadamente específicas (em vez disso, use categorias de superclasse, para evitar problemas de manutenção no futuro).

RESUMO DA LINGUAGEM BÁSICA

Parabéns! Isto conclui seu estudo da linguagem de programação Python básica. Se você chegou até aqui, pode se considerar um Programador de Python Oficial (e deve se sentir à vontade para incluir o Python em seu currículo, na próxima vez que mexer nele). Você já viu praticamente tudo que há para ver na linguagem em si – tudo com muito mais profundidade do que muitos programadores de Python profissionais. Da Parte II até a Parte VII do livro, você estudou os tipos internos, instruções e exceções, assim como as ferramentas usadas para construir unidades de programa maiores – funções, módulos e classes – e explorou problemas de projeto, POO, arquitetura de programa e muito mais.

vimento no Python e no domínio público. Você já viu algumas delas em ação e já falamos sobre outras. Para ajudá-lo em seu caminho, aqui está um resumo de algumas das ferramentas mais comumente usadas nesse domínio, muitas das quais você já viu:

PyDoc e docstrings

Apresentamos a função `help` da PyDoc e interfaces HTML no Capítulo 11. A PyDoc fornece um sistema de documentação para seus módulos e objetos, e se integra com o recurso docstrings do Python. A PyDoc é uma parte padrão do sistema Python. Consulte o manual da biblioteca para ver mais detalhes. Consulte também as dicas de fonte de documentação listadas no Capítulo 11, para ver recursos de informação sobre o Python em geral.

PyChecker

Como o Python é uma linguagem muito dinâmica, alguns erros de programação não são relatados até que seu programa seja executado (por exemplo, os erros de sintaxe são capturados quando um arquivo é executado ou importado). Esse não é um grande inconveniente – assim como na maioria das linguagens, significa apenas que você precisa testar seu código Python antes de distribuí-lo. Além disso, a natureza dinâmica, as mensagens de erro automáticas e o modelo de exceção do Python tornam mais fácil e mais rápido encontrar e corrigir erros do que em algumas linguagens (ao contrário da linguagem C, o Python não falha na ocorrência de erros).

Entretanto, o sistema PyChecker fornece suporte para capturar antecipadamente um grande conjunto de erros comuns, antes que seu script seja executado. Ele tem funções semelhantes ao programa "lint" para desenvolvimento em C. Alguns grupos que usam Python executam seus códigos por meio do PyChecker antes de testar ou distribuir, para capturar todos os problemas furtivos em potencial. Na verdade, a biblioteca padrão do Python é regularmente executada por meio do PyChecker, antes do lançamento. O PyChecker é um pacote de outro fornecedor; você o encontra no endereço *http://www.python.org* ou no site da Web Vaults of Parnassus.

PyUnit (também conhecido como unittest)

Na Parte V, demonstramos como se faz para adicionar código de auto-teste em arquivos do Python, usando o truque `__name__ == '__main__'` no final do arquivo. Para propósitos de testes mais avançados, o Python vem com duas ferramentas de suporte para teste. A primeira, PyUnit (chamada de `unittest` no manual da biblioteca), fornece um modelo de classe orientado a objetos para especificar e personalizar casos de teste e resultados esperados. Ela imita o modelo JUnit da linguagem Java. Trata-se de um sistema baseado em classes. Consulte o manual da biblioteca do Python para ver os detalhes.

Doctest

O módulo da biblioteca padrão `doctest` fornece uma segunda estratégia (mais simples) para testes de regressão. Ele é baseado no recurso docstrings do Python. A grosso modo, sob o `doctest`, você recorta e cola um log de uma sessão de teste interativa para as docstrings de seus arquivos-fonte. Então, o doctest extrai suas docstrings, analisa os casos de teste e os resultados, e executa os testes novamente para verificar os resultados esperados. A operação do doctest pode ser personalizada de várias maneiras. Consulte o manual da biblioteca para ver mais informações sobre sua operação.

IDEs

Discutimos os IDEs para Python no Capítulo 3. Os IDEs, assim como o IDLE, fornecem um ambiente gráfico para editar, executar, depurar e navegar em seus programas em Python. Alguns IDEs avançados, como o Komodo, suportam tarefas de desenvolvimento

digo de byte *.pyo* otimizados, gerados e executados com o flag de linha de comando `-O` do Python, discutido no Capítulo 15. Como isso oferece um ganho de desempenho muito modesto, não é comumente usado. Como último recurso, você também pode mover partes de seu programa para uma linguagem compilada, como a C, para aumentar o desempenho. Consulte o livro *Programming Python* e os manuais padrão do Python para ver mais informações sobre as extensões da linguagem C. Em geral, a velocidade do Python também melhora com o passar do tempo; portanto, migre para a versão mais recente, quando possível (a versão 2.3 foi cronometrada e resultou de 15 a 20% mais rápida do que a 2.2).

Outras dicas para projetos maiores

Finalmente, conhecemos uma variedade de recursos da linguagem que tendem a se tornar mais úteis quando você começa a desenvolver projetos maiores. Dentre eles estão: pacotes de módulo (Capítulo 17), exceções baseadas em classe (Capítulo 25), atributos pseudo-privados de classe (Capítulo 23), strings de documentação (Capítulo 11), arquivos de configuração de caminho de módulo (Capítulo 15), ocultação de nomes de `from*` com listas `__all__` e nomes de estilo `_X` (Capítulo 18), adição de código de auto-teste com o truque `__name__ == '__main__'` (Capítulo 18), uso de regras de projeto comuns para funções e módulos (Capítulos 5 e 6) etc.

Para ver mais ferramentas de desenvolvimento em larga escala do Python, disponíveis no domínio público, navegue nas páginas do site da Web Vaults of Parnassus.

EXERCÍCIOS DA PARTE VII

Como estamos no final da abordagem da linguagem básica, vamos trabalhar em alguns exercícios curtos sobre exceção, para dar a você uma chance de praticar os fundamentos. As exceções representam uma ferramenta realmente simples; portanto, se você fizer os exercícios, as terá dominado.

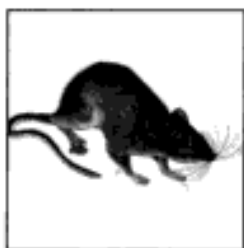
1. *try/except*. Escreva uma função chamada `oops` que lance explicitamente uma exceção `IndexError`, quando chamada. Em seguida, escreva outra função que chame `oops` dentro de uma instrução `try/except` para capturar o erro. O que acontece se você altera `oops` para lançar `KeyError`, em vez de `IndexError`? De onde vêm os nomes `KeyError` e `IndexError`? (Dica: lembre-se de que todos os nomes não qualificados são provenientes de um de quatro escopos, pela regra LEGB.)
2. *Objetos exceção e listas*. Altere a função `oops` que acabou de escrever, para lançar uma exceção que você mesmo define, chamada `MyError`, e passe um item de dados extra junto com a exceção. Você pode identificar sua exceção com uma string ou com uma classe. Em seguida, estenda a instrução `try` na função capturadora para capturar essa exceção e seus dados, além de `IndexError`, e imprimir o item de dados extra. Finalmente, se você usou uma string para sua exceção, volte e altere-a para uma instância de classe. Agora, o que retorna como dados extras para a rotina de tratamento?
3. *Tratamento de erro*. Escreva uma função chamada `safe(func, *args)` que execute qualquer função usando `apply`, capture qualquer exceção lançada enquanto a função é executada e imprima a exceção usando os atributos `exc_type e exc_value` no módulo `sys`. Em seguida, use sua função `safe` para executar a função `oops` que você escreveu nos exercícios 1 e/ou 2. Coloque `safe` em um arquivo de módulo chamado *tools.oy* e passe-a para a função `oops` interativamente. Que tipo de mensagens de erro você recebe? Finalmente, expanda `safe` para imprimir também um rastreamento de pilha do Python quando ocorrer um erro, chamando a função interna `print_exc()` no módulo padrão `traceback` (consulte o manual de referência da biblioteca Python para ver os detalhes).

VIII

As Camadas Externas

Até aqui, abordamos os fundamentos da linguagem Python. Com esse conhecimento, você conseguirá ler praticamente todo código Python, com poucas surpresas relacionadas à linguagem. Entretanto, conforme todo mundo que já trabalhou com programas sabe, entender a sintaxe de uma linguagem não garante um entendimento claro e fácil de um programa, mesmo que ele seja bem escrito. Na verdade, saber quais ferramentas estão sendo usadas – sejam elas funções simples, pacotes coerentes ou mesmo modelos complexos – é a etapa importante entre um entendimento teórico de uma linguagem e o domínio prático e efetivo de um sistema.

Como você pode fazer essa transição? Nenhuma quantidade de leitura de revistas especializadas em marcenaria transformará um iniciante em um perito em trabalhos com madeira. Para que isso aconteça, você precisa ter talento, é claro, mas também precisa passar vários anos examinando móveis, desmontando-os, construindo peças novas e aprendendo com seus erros e com o sucesso dos outros. O mesmo vale para a programação. A função dos livros-texto é oferecer um panorama dos tipos de problemas e das soluções apropriadas, para apontar alguns dos truques básicos do ramo e motivar o iniciante frustrado, mostrando algum trabalho bem feito de outras pessoas. A Parte VIII apresenta uma visão diferente do panorama do Python em cada capítulo e fornece muitas indicações de outras fontes de informação.

27

Tarefas Comuns no Python

Neste ponto do livro, você já ficou exposto a um levantamento bastante completo dos aspectos mais formais da linguagem (a sintaxe, os tipos de dados etc.). Neste capítulo, vamos "sair da sala de aula", vendo um conjunto de tarefas básicas de computação e examinando como os programadores de Python normalmente as resolvem, na esperança de ajudá-lo a fundamentar o conhecimento teórico com resultados concretos.

Os programadores de Python não gostam de reinventar a roda, quando já têm acesso a rodas excelentes e redondas em sua garagem. Assim, o conteúdo mais importante deste capítulo é a descrição de ferramentas selecionadas que constituem a biblioteca padrão do Python – funções internas, módulos de biblioteca e suas funções e classes mais úteis. Embora seja mais provável que você não vá usar tudo isso em um só programa, nenhum programa útil deixa de usá-las. Assim como o Python fornece um tipo de objeto lista porque as manipulações de seqüência ocorrem em todos os contextos de programação, a biblioteca fornece um conjunto de módulos que serão úteis muitas vezes. Antes de projetar e escrever qualquer código útil de modo geral, verifique se já existe um módulo semelhante. Se ele fizer parte da biblioteca padrão do Python, você poderá ter certeza de que já foi bastante testado; melhor ainda, outras pessoas estão empenhadas em corrigir todos os erros restantes, gratuitamente.

O objetivo deste capítulo é mostrar muitas ferramentas diferentes para que você saiba que elas existem, em vez de ensiná-lo tudo que precisa saber para utilizá-las. Existem fontes de conhecimento complementares muito boas, para quando você tiver terminado de ler este livro. Se você quiser explorar melhor a biblioteca padrão, a referência definitiva é a Python Library Reference (Referência da Biblioteca do Python), atualmente com mais de 600 páginas. Ela é a acompanhante ideal para este livro, pois proporciona o complemento para o qual não temos espaço e, estando disponível on-line, é a descrição mais atualizada do conjunto de ferramentas padrão do Python. Três outros livros da O'Reilly fornecem informações adicionais excelentes: o livro *Python Pocket Reference*, escrito por Mark Lutz, aborda os módulos mais importantes da biblioteca padrão, junto com a sintaxe e funções internas, em forma compacta; o livro *Python Standard Library*, de Frederik Lundh, empreende a formidável tarefa de fornecer documentação adicional para cada módulo da biblioteca padrão, assim como um exemplo de programa mostrando como usar cada módulo; e, finalmente, o livro *Python in a Nutshell*, de Alex Martelli, fornece uma descrição completa e, ainda assim, eminentemente fácil de ler e concisa,

da linguagem e da biblioteca padrão. Conforme veremos na seção "Explorando por sua própria conta", o Python também vem com ferramentas que facilitam o auto-aprendizado.

Assim como não podemos abordar cada módulo padrão, o conjunto de tarefas abordadas neste capítulo é necessariamente limitado. Se você quiser mais, consulte o livro *Python Cookbook* (O'Reilly), editado por David Ascher e Alex Martelli. Esse livro de receitas aborda muitos dos mesmos domínios de problema que mencionamos aqui, mas com profundidade muito maior e com muito mais discussões. Promovendo o conhecimento coletivo da comunidade Python, esse livro fornece um levantamento muito mais amplo e rico das estratégias do Python para tarefas comuns.

Este capítulo limita-se às ferramentas disponíveis como parte das distribuições padrão do Python. Os dois próximos capítulos abrangem o assunto dos módulos e bibliotecas de outros fornecedores, pois muitos desses módulos podem ser valiosos para o programador de Python.

Este capítulo começa abordando as tarefas comuns que se aplicam aos conceitos de programação fundamentais – tipos, estruturas de dados, strings –, passando para assuntos de nível conceitual mais alto, como arquivos e diretórios, operações relacionadas à Internet e execução de processos, antes de terminar com algumas tarefas de programação como testes, depuração e traçado de perfil.

Explorando por sua própria conta

Antes de nos aprofundarmos nas tarefas específicas, devemos falar brevemente sobre a auto-exploração. Não fomos exaustivos na abordagem dos atributos de objeto nem do conteúdo dos módulos, para focalizarmos os aspectos mais importantes dos objetos sob discussão. Se você estiver curioso sobre o que omitimos, pode pesquisar a Library Reference ou remexer no interpretador interativo do Python, conforme mostrado nesta seção.

A função interna `dir` retorna uma lista de todos os atributos de um objeto e, junto com a função interna `type`, proporciona uma maneira excelente de aprender sobre os objetos que você está manipulando. Por exemplo:

```
>>> dir([])                      # Quais são os atributos das listas?
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__', '__
getitem__'
'__getslice__', '__gt__', __hash__', '__iadd__', __imul__', __init__',
'__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__repr__', '__rmul__', '__setattr__', '__setitem__', 'setslice__', '__str__',
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

O que isso diz é que o objeto lista vazia tem alguns métodos: `append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, `sort`, e muitos "métodos especiais" que começam com um sublinhado (_) ou dois (__). Eles são usados nos bastidores pelo Python, ao executar operações como `+`. Como esses métodos especiais não são necessários com muita freqüência, escreveremos uma função utilitária simples que não os exibe:

```
>>> def mydir(obj):
...     orig_dir = dir(obj)
```

```
...     return [item for item in orig_dir if not item.startswith('_')]
>>>
```

Usar essa nova função na mesma lista vazia produz:

```
>>> mydir([])                          # Quais são os atributos de listas?
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Você pode, então, explorar qualquer objeto do Python:

```
>>> mydir(())                          # Quais são os atributos de tuplas?
[]                                     # Nota: não atributos "normais"
>>> import sys                         # Quais são os atributos de arquivos?
>>> mydir(sys.stdin)                   # Quais são os atributos de arquivos?
['close', 'closed', 'fileno', 'flush', 'isatty', 'mode', 'name', 'read',
'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell',
'truncate', 'writelines', 'xreadlines']
>>> mydir(sys)                         # Módulos também são objetos.
['argv', 'builtin_module_names', 'byteorder', 'copyright', 'displayhook',
'dllhandle', 'exc_info', 'exc_type', 'excepthook', 'exec_prefix', 'executable',
'exit', 'getdefaultencoding', 'getrecursionlimit', 'getrefcount',
'hexversion',
'last_traceback', 'last_type', 'last_value', 'maxint', 'maxunicode',
'modules',
'path', 'platform', 'prefix', 'ps1', ps2', 'setcheckinterval', 'setprofile',
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
'version_info', 'warnoptions', 'winver']
>>> type(sys.version)                  # Que tipo de coisa é 'version'?
<type 'string'>
>>> print repr(sys.version)            # Qual é o valor dessa string?
'2.3a1 (#38, Dec 31 2002, 17:53:59) [MSC v.1200 32 bit (Intel)]'
```

As versões recentes do Python também contêm uma função interna que é muito útil para os iniciantes, chamada (de forma muito apropriada) `help`:

```
>>> help(sys)
help on built-in module sys:
NAME
    sys

FILE
    (built-in)

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the
    interpreter.

Dynamic objects:

argv-command line arguments; argv[0] is the script pathname if known
path-module search path; path[0] is the script directory, else ' '
modules-dictionary of loaded modules
displayhook-called to show results in an interactive session
excepthook-called to handle any uncaught exception other than
SystemExit
```

```
To customize printing in an interactive session or to install a
custom top-level exception handler, assign other functions to replace
these.
...
```

Isso é bastante para um sistema de ajuda on-line. Recomendamos que você comece primeiro neste estado "modal", apenas digitando `help()`. Daí em diante, qualquer string que você digitar produzirá sua documentação. Digite `quit` para sair do modo de ajuda.

```
>>> help()
Welcome to Python 2.2! This is the online help utility
...
help> socket
help on module socket:

NAME
    socket

FILE
    c:\python22\lib\socket.py

DESCRIPTION
    This module provides socket operations and some related functions.
    On Unix, it supports IP (Internet Protocol) and Unix domain sockets.
    On other systems, it only supports IP. Functions specific for a
    socket are avaliable as methods of the socket object.

    Functions:

    socket() -- create a new socket object
    fromfd() -- created a socket object from an open file descriptor [*]
    gethostname() -- return the current hostname
    gethostbyname() -- map a hostname to its IP number
    gethostbyaddr() -- map an IP number or hostname to DNS info
    getservbyname() -- map a service name and a protocol name to a port
    ...
help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

and                 elif            global     or
assert              else            if         pass
break               except          import     print
class               exec            in         raise
continue            finally         is         return
def                 for             lambda     try
del                 from            not        while
help> topics

Here is a list of available topics. Enter any topic name to get more help.

ASSERTION             DEBUGGING             LITERALS        SEQUENCEMETHODS1
ASSIGNMENT            DELETION              LOOPING         SEQUENCEMETHODS2
ATTRIBUTEMETHODS      DICTIONARIES          MAPPINGMETHODS  SEQUENCES
ATTRIBUTES            DICTIONARYLITERALS    MAPPINGS        SHIFTING
AUGMENTEDASSIGNMENT   ELLIPSIS              METHODS         SLICINGS
BACKQUOTES            EXCEPTIONS            MODULES         SPECIALATTRIBUTES
BASICMETHODS          EXECUTION             NAMESPACES      SPECIALIDENTIFIERS
BINARY                EXPRESSIONS           NONE            SPECIALMETHODS
```

```
BITWISE             FILES               NUMBERMETHODS    STRINGMETHODS
BOOLEAN             FLOAT               NUMBERS          STRINGS
CALLABLEMETHODS     FORMATTING          OBJECTS          SUBSCRIPTS
CALLS               FRAMEOBJECTS        OPERATORS        TRACEBACKS
CLASSES             FRAMES              PACKAGES         TRUTHVALUE
CODEOBJECTS         FUNCTIONS           POWER            TUPLELITERALS
COERCIONS           IDENTIFIERS         PRECEDENCE       TUPLES
COMPARISON          IMPORTING           PRINTING         TYPEOBJECTS
COMPLEX             INTERGER            PRIVATENAMES     TYPES
CONDITIONAL         LISTLITERALS        RETURNING        UNARY
CONVERSIONS         LISTS               SCOPING          UNICODE
help> TYPES
  3.2 The standard type hierarchy

  Below is a list of the types that are built into Python. Extension
  modules written in C can define additional types. Future versions of
  Python may add types to the type hierarchy (e.g., rational numbers,
  efficiently stored arrays of integers, etc.).
  ...

help> quit
>>>
```

CONVERSÕES, NÚMEROS E COMPARAÇÕES

Quando abordamos os tipos de dados, um dos problemas comuns ao tratarmos com qualquer sistema de tipo é como alguém converte de um tipo para outro. Essas conversões acontecem em uma miríade de contextos – leitura de números de um arquivo de texto, cálculo de média de valores inteiros, interface com funções que esperam tipos diferentes do restante de uma aplicação etc.

Nos capítulos anteriores, vimos que é possível criar uma string a partir de um objeto que não é uma string, simplesmente passando esse objeto para o construtor de string `str`. Analogamente, `unicode` converte qualquer objeto em sua forma de string para Unicode e a retorna.*

Além das funções de criação de string, vimos `list` e `tuple`, que recebem seqüências e retornam versões de lista e tupla, respectivamente. `int`, `complex`, `float` e `long` recebem qualquer número e o convertem em seus respectivos tipos. `int`, `long` e `float` têm recursos adicionais que podem ser confusos. Primeiramente, `int` e `long` truncam seus argumentos numéricos, se necessário, para efetuar a operação, perdendo com isso informações e realizando uma conversão que pode não ser a que você deseja (a função interna `round` arredonda números da maneira padrão e retorna um valor em ponto flutuante). Segundo, `int`, `long` e `float` também podem converter strings para seus respectivos tipos, desde que as strings sejam literais inteiras (longas ou de ponto flutuante) válidas. As literais são strings de texto que são convertidas em números no início do processo de compilação do Python. Assim, a string 1244 em seu arquivo de programa em Python (que é necessariamente uma string) é uma literal inteira válida, mas `def foo():`, não.

```
>>> int(1.0), int(1.4), int(1.9), round(1.9), int(round(1.9))
(1, 1, 1, 2.0, 2)
>>> int("1")
1
>>> int("1.2")                              # Isso não funciona.
```

* Como não vamos fazer subclasses de tipos internos neste capítulo, não faz diferença se essas chamadas de conversão são funções (o que elas eram até versões recentes do Python) ou criadoras de classe (o que elas são no Python 2.2 e posteriores) – de qualquer modo, elas recebem objetos como entrada e retornam novos objetos do tipo apropriado (supondo que a conversão específica seja permitida). Nesta seção, vamos nos referir a elas como funções, por questão de conveniência.

```
# precisa ser "unida" em uma str.
>>> map(chr, (83, 112, 97, 109, 33))
['S', 'p', 'a', 'm', '!']
# Também pode ser escrito usando abrangências de lista
>>> [chr(x) for x in (83, 112, 97, 109, 33)]
['S', 'p', 'a', 'm', '!']
>>> ''.join ([chr(x) for x in (83, 112, 97, 109, 33)])
'Spam!'
```

A função interna `cmp` retorna um inteiro negativo, 0 ou um inteiro positivo, dependendo de seu primeiro argumento ser menor, igual ou maior que o segundo. Vale a pena enfatizar que a função `cmp` funciona com mais do que apenas números; ela compara caracteres usando seus valores ASCII e seqüências são comparadas pela comparação de seus elementos. As comparações podem lançar exceções; portanto, não é garantido que a função de comparação funcione em todos os objetos, mas todas as comparações razoáveis funcionarão.* O processo de comparação usado por `cmp` é igual àquele usado pelo método de listas `sort`. Ele também é usado pelas funções internas `min` e `max`, que retornam o menor e o maior elementos dos objetos com que são chamadas, lidando com seqüências razoavelmente:

```
>>> min("pif", "paf", "pof")          # Quando chamada com vários argumentos,
'paf'                                 # retorna o que for apropriado.
>>> min("ZELDA!"), max("ZELDA!")      # Quando chamada com uma seqüência,
'!', 'Z'                              # retorna o elemento min/máx dela.
```

A Tabela 27-1 resume as funções internas que tratam com conversões de tipo. Muitas delas também podem ser chamadas sem nenhum argumento para retornar um valor falso; por exemplo, `str()` retorna a string vazia.

Tabela 27-1 Funções internas de conversão de tipo

Nome da função	Comportamento

`str(string)`
`unicode(string)`
	Retorna a representação de string de qualquer objeto:
	`>>> str(dir())`
	`"['__builtins__', '__doc__', '__name__']"`
	`>>> unicode('tomato')`
	`u' tomato')`

`list(seq`	Retorna a versão de lista de uma seqüência:
	`)>>> list("tomato")`
	`['t','o', 'm', 'a', 't', 'o']`
	`>>> list((1,2,3))`
	`[1, 2, 3]`

`tuple(seq)`	Retorna a versão de tupla de uma seqüência:
	`>>> tuple("tomato")`
	`('t', 'o', 'm', 'a', 't', 'o')`
	`>>> tuple([0])`
	`(0,)`

* Por diversos motivos, principalmente históricos, mesmo comparações não razoáveis (1 > "2") produzirão um valor.

MANIPULANDO STRINGS

A ampla maioria dos programas efetua operações de string. Abordamos a maioria das propriedades e variantes dos objetos string no Capítulo 5, mas existem duas áreas em que não mexemos até agora: o módulo string e as expressões regulares. Conforme veremos, a primeira é uma nota simples e principalmente histórica, enquanto a segunda é complexa e poderosa.

O módulo string

O módulo `string` é um pouco uma anomalia histórica. Se o Python estivesse sendo projetado hoje, o módulo `string` não existiria – ele é principalmente um vestígio de uma era menos civilizada, antes que tudo fosse objeto de primeira classe. Atualmente, os objetos `string` têm métodos como `split` e `join`, que substituem as funções que ainda são definidas no módulo `string`. Mas o módulo string define uma função conveniente, `maketrans`, usada para executar automaticamente operações de "mapeamento" de string no método `translate` de objetos string. O par `maketrans/translate` é útil quando você quer transformar vários caracteres de uma vez em uma string. Por exemplo, se você quiser substituir todas as ocorrências do caractere de espaço por um sublinhado, altere os sublinhados para sinais de subtração e os sinais de subtração para sinais de adição. Na verdade, fazer isso com operações `.replace()` repetidas é muito complicado, mas com `maketrans` é trivial:

```
>>> import string
>>> conversion = string.maketrans(" _-", "_-+")
>>> input_string = "This is a two_part - one_part"
>>> input_string.translate(conversion)
'This_is_a_two-part_+_one-part'
```

Além disso, o módulo `string` define algumas constantes úteis que ainda não foram implementadas como atributos de string. Elas aparecem na Tabela 27-2.

Tabela 27-2 Constantes do módulo string

Nome da constante	Valor
`digits`	`'0123456789'`
`octdigits`	`'01234567'`
`hexdigits`	`'0123456789abcdefABCDEF'`
`lowercase`	`'abcdefghijklmnopqrstuvwxyz'`
`uppercase`	`'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`
`Letters`	`lowercase + uppercase`
`whitespace`	`'\t\n\r\v'` **(todos os caracteres de espaço em branco)**

As constantes da Tabela 27-2 são úteis para testar se caracteres específicos satisfazem um critério – por exemplo, `x in string.whitespace` retorna verdadeiro somente se `x` é um dos caracteres de espaço em branco. Note que os valores dados na tabela nem sempre são os que você encontrará – por exemplo, a definição de `'uppercase'` depende da localidade: se você estiver executando um sistema operacional francês, `string.lowercase` incluirá `ç` e `ê`.

Correspondências de strings complicadas com expressões regulares

Se as strings e seus métodos não são suficientes (e elas ficam complicadas em muitos casos de uso perfeitamente normais), o Python fornece uma ferramenta de processamento de strings especializada, na forma de um mecanismo de expressão regular.

As expressões regulares são strings que permitem definir regras de correspondência e substituição de padrão complicadas para strings. A sintaxe das expressões regulares enfatiza a notação compacta sobre o valor menmônico. Por exemplo, o caractere simples . significa "corresponder a qualquer caractere simples". O caractere + significa "um ou mais do que acabou de me preceder". A Tabela 27-3 lista alguns dos símbolos de expressão regular mais comumente usados e seus significados em português. Descrever o conjunto completo de sinais de expressão regular e seus significados exigiria muitas páginas – em vez disso, abordaremos um caso de uso simples e mostraremos como resolver o problema usando expressões regulares.

Tabela 27-3 Elementos comuns da sintaxe de expressão regular

Caractere especial	Significado
`.`	Corresponde a qualquer caractere, exceto o de nova linha, por padrão
`^`	Corresponde ao início da string
`$`	Corresponde ao final da string
`*`	"Qualquer número de ocorrências do que acabou de me preceder"
`+`	"Uma ou mais ocorrências do que acabou de me preceder"
`|`	"Uma das coisas antes de mim ou a coisa depois de mim"
`\w`	Corresponde a qualquer caractere alfanumérico
`\d`	Corresponde a qualquer dígito decimal
`tomato`	Corresponde à string tomato

Um problema real de expressão regular

Suponha que você precise escrever um programa para substituir as strings "green pepper" e "red pepper" por "bell pepper", se e somente se elas ocorrerem juntas em um parágrafo, antes da palavra "salad", e não se elas forem seguidas (sem nenhum espaço) pela string "corn". Embora os requisitos específicos sejam bobos, o tipo geral (substituição condicional de trechos de texto com base em restrições contextuais específicas) é surpreendentemente comum na computação. Explicaremos cada etapa do programa que resolve essa tarefa.

Suponha que o arquivo que você precisa processar se chame *pepper.txt*. Aqui está um exemplo de tal arquivo:

```
This is a paragraph that mentions bell peppers multiple times. For one, here
is a red pepper and dried tomato salad recipe. I don't like to use green
peppers in my salads as much because they have a harsher flavor.

This second paragraph mentions red peppers and green peppers but not the "s"
word (s-a-l-a-d), so no bells should show up.

This third paragraph mentions red peppercorns and green peppercorns, which
aren't vegetables but spices (by the way, bell peppers really aren't peppers,
they're chilies, but would you rather have a good cook or a good botanist
prepare your salad?).
```

A primeira tarefa é abrir o arquivo e ler o texto:

```
file = open('pepper.txt')
text = file.read()
```

Lemos o texto inteiro de uma vez e evitamos sua divisão em linhas, pois vamos supor que os parágrafos sejam definidos por dois caracteres de nova linha consecutivos. É fácil fazer isso, usando a função `split` do módulo string:

`+` significa “uma ou mais ocorrências do que vier antes de mim”; portanto, juntos, `\s+` significam “um ou mais caracteres de espaço em branco”. Então, pepper significa apenas a string `'pepper'`. `(?!corn)` impede as correspondências de “padrões que tenham `'corn'` neste ponto”; portanto, impedimos a correspondência em `'peppercorn'`. Finalmente, `(?=.*salad)` diz que para o padrão corresponder, ele precisa ser seguido por qualquer número de caracteres arbitrários (é isso que `.*` significa), seguidos da palavra `salad`. Os caracteres `?=` especificam que, embora o padrão deva determinar se a correspondência ocorre, ela não deve ser “consumida” pelo processo de correspondência; esse é um ponto sutil que não abordaremos em detalhes aqui. Nesse ponto, definimos a correspondência de padrão para a substring.

Agora, note que existem dois parênteses – um antes de `\s+` e o último. O que eles fazem é definir um “grupo”, que começa após a palavra `red` ou `green` e vai até o final do padrão. Vamos usar esse grupo na próxima operação, a substituição real. Os três flags são unidos pelo símbolo | (a operação “ou” com reconhecimento de bit) para formar o segundo argumento de `re.compile`. Eles especificam tipos de correspondência de padrão. O primeiro `re.IGNORECASE`, diz que as comparações de texto devem ignorar o fato de o texto e a correspondência terem caixas semelhantes ou diferentes. O segundo, `re.DOTALL`, especifica que o caractere . deve corresponder a qualquer caractere, incluindo o caractere de nova linha (esse não é o comportamento padrão). O terceiro, `re.VERBOSE`, nos permite inserir caracteres de nova linha extras e comentários `#` na expressão regular, tornando-a mais fácil de ler e entender. Poderíamos ter escrito a instrução de forma mais compacta, como:

```
matchstr = re.compile(r"\b(red|green)(\s+pepper(?!corn)(?=.*salad))", re.I | re.S)
```

A operação de substituição real é efetuada com a linha:

```
fixed_paragraph = matchstr.sub(r'bell\2', paragraph)
```

Estamos chamando o método `sub` do objeto `matchstr`. Esse é um objeto expressão regular compilada, significando que parte do processamento da expressão já foi realizado (neste caso, fora do loop), acelerando assim a execução total do programa. Usamos uma string bruta novamente, para escrever o primeiro argumento do método. `\2` é uma referência para o grupo 2 na expressão regular – em nosso caso, tudo que começa com um espaço em branco, seguido de `'pepper'`, até (e incluindo) a palavra `'salad'`. Portanto, essa linha significa “substituir as ocorrências da substring da correspondência pela string que é `'bell'`, seguida do que começar com um espaço em branco, seguido de `'pepper'` e até o final da string da correspondência, por toda a string paragraph”.

Então, isso funciona? O arquivo *pepper.txt* tinha três parágrafos: o primeiro satisfez duas vezes os requisitos da correspondência; o segundo, não, pois não mencionou a palavra `'salad'`; e o terceiro, não, pois as palavras `'red'` e `'green'` estão antes de peppercorn e não de pepper. Conforme se supunha, nosso programa (salvo em um arquivo chamado *pepper.py*) modifica apenas o primeiro parágrafo:

```
/home/David/book$ python pepper.py

This is a paragraph that mentions bell peppers multiple times. For one, here
is a bell pepper and dried tomato salad recipe. I don't like to use bell
peppers in my salads as much because they have a harsher flavor.

This second paragraph mentions red peppers and green peppers but not the "s"
word (s-a-l-a-d), so no bells should show up.

This third paragraph mentions red peppercorns and green peppercorns, which
aren't vegetables but spices (by the way, bell peppers really aren't peppers,
```

```
they're chilies, but would you rather have a good cook or a good botanist
prepare your salad?).
```

Esse exemplo, embora artificial, mostra como as expressões regulares podem expressar regras de correspondência de forma compacta. Se esse tipo de problema ocorre freqüentemente em seu ramo de atividade, dominar as expressões regulares pode valer o investimento de tempo e esforço.

Uma abordagem mais completa das expressões regulares está fora dos objetivos deste livro. Jeffrey Friedl fornece uma abordagem excelente sobre expressões regulares em seu livro *Mastering Regular Expressions* (O'Reilly). Esse livro é obrigatório para quem estiver realizando processamento de textos sério. Para o usuário ocasional, as descrições da Library Reference e do livro *Python in a Nutshell* resolvem o problema, na maioria das vezes. Certifique-se de usar o módulo `re` e não os módulos `regex` ou `regsub`, que estão obsoletos (eles provavelmente não aparecerão em uma versão posterior do Python):

```
>>> import regex
__main__:1: DeprecationWarning: the regex module is deprecated; please use
the remodule
```

MANIPULAÇÕES DE ESTRUTURA DE DADOS

Uma das melhores características do Python é que ele fornece os tipos internos lista, tupla e dicionário. Eles são tão flexíveis e fáceis de usar que, uma vez que você tenha se acostumado com eles, passará a utilizá-los automaticamente. Embora tenhamos abordado todas as operações sobre cada estrutura de dados, quando as apresentamos, agora é um bom momento para ver as tarefas que podem ser aplicadas a todas elas, como o modo de fazer cópias, ordenar objetos, tornar seqüências aleatórias etc. Muitas funções e algoritmos (procedimentos teóricos que descrevem como implementar uma tarefa complexa em termos de tarefas básicas mais simples) são projetados para funcionar independente do tipo de dados que estão sendo manipulados. Portanto, é útil saber como se faz coisas genéricas para todos os tipos de dados.

Fazendo cópias

Fazer cópias de objetos é uma tarefa razoável em muitos contextos de programação. Freqüentemente, o único tipo de cópia necessária é apenas outra referência para um objeto, como em:

```
x = 'tomato'
y = x                          # agora y é 'tomato'.
x = x + ' and cucumber'        # agora x é 'tomato and cucumber',
                               # mas y fica inalterado.
```

Devido ao esquema de gerenciamento de referências do Python, a instrução `a = b` não faz uma cópia do objeto referenciado por `b`; em vez disso, ela cria uma nova referência para esse mesmo objeto. Quando o objeto que está sendo copiado é imutável (por exemplo, uma string), não há nenhuma diferença real. Entretanto, ao se tratar com objetos mutáveis, como listas e dicionários, às vezes é necessária uma nova cópia do objeto e não apenas uma referência compartilhada. O modo de fazer isso depende do tipo do objeto em questão. A maneira mais simples de fazer uma cópia é usando os construtores `list()` ou `tuple()`:

```
newList = list(myList)
newTuple = tuple(myTuple)
```

Em oposição à maneira mais simples, o modo mais comum de fazer cópias de seqüências, como listas e tuplas, é um tanto estranho. Se `myList` é uma lista, então, para fazer uma cópia dela, você pode usar:

```
newList = myList[:]
```

que pode ser lido como "fracione do início ao fim", pois o índice padrão do início de um fracionamento é o início da seqüência (0) e o índice padrão do final de um fracionamento é o final da seqüência (consulte o Capítulo 3). Como as tuplas suportam a mesma operação de fracionamento que as listas, essa mesma técnica também pode ser aplicada a elas, exceto que, se `x` é uma tupla, então `x[:]` é o mesmo objeto que `x`, pois as tuplas são imutáveis. Por outro lado, os dicionários não suportam fracionamento. Para fazer uma cópia de um dicionário `myDict`, você pode usar:

```
newDict = myDict.copy()
```

ou o construtor `dict()`:

```
newDict = dict(myDict)
```

Para um tipo diferente de cópia, se você tiver um dicionário `oneDict` e quiser atualizá-lo com o conteúdo de um dicionário `otherDict` diferente, basta digitar `oneDict.update(otherDict)`. Isso é o equivalente de:

```
for key in otherDict.keys():
    oneDict[key] = otherDict[key]
```

Se `oneDict` compartilha algumas das chaves com `otherDict` antes da operação `update()`, os antigos valores associados às chaves em `oneDict` são destruídos pela atualização. Talvez você queira fazer isso mesmo (normalmente, você quer). Se não, o certo a fazer pode ser lançar uma exceção. Para isso, fazemos uma cópia de um dicionário e depois examinamos cada entrada no segundo. Se encontrarmos chaves compartilhadas, lançamos uma exceção; caso contrário, apenas adicionamos o mapeamento chave-valor no novo dicionário.

```
def mergeWithoutOverlap(oneDict, otherDict):
    newDict = oneDict.copy()
    for key in otherDict:
        if key in oneDict:
            raise ValueError, "the two dictionaries share keys!"
        newDict[key] = otherDict[key]
    return newDict
```

ou, como alternativa, combinamos os valores dos dois dicionários, por exemplo, com uma tupla. Usando a mesma lógica de `mergeWithoutOverlap`, mas combinando os valores, em vez de lançar uma exceção:

```
def mergeWithoutOverlap(oneDict, otherDict):
    newDict = oneDict.copy()
    for key in otherDict:
        if key in oneDict:
            newDict[key] = oneDict[key], otherDict[key]
        else:
            newDict[key] = otherDict[key]
    return NewDict
```

Para ilustrar as diferenças entre os três algoritmos anteriores, considere os dois dicionários a seguir:

```
phoneBook1 = {'michael': '555-1212', 'mark': '554-1121', 'emily': '556-0091'}
phoneBook2 = {'latoya': '555-1255', 'emily': '667-1234'}
```

Se possivelmente `phoneBook1` está desatualizado e `phoneBook2` está mais atualizado, mas é menos completo, a utilização correta provavelmente é `phoneBook1.update(phoneBook2)`. Se os

dois dicionários phoneBook supostamente possuem conjuntos de chaves não sobrepostas, o uso de `newBook = mergeWithoutOverlap(phoneBook1, phoneBook2)` permite que você saiba se essa suposição está errada. Finalmente, se um deles é um conjunto de números de telefones residenciais e o outro é um conjunto de números de telefones comerciais, as chances são de que `newBook = mergeWithoutOverlap(phoneBook1, phoneBook2)` é o que você quer, desde que o código subseqüente que usar `newBook` possa lidar com o fato de que `newBook['emily']` é a tupla `('556-0091', '667-1234')`.

O módulo copy

Os truques `[:]` e `.copy()` fornecerão as cópias para você em 90% dos casos. Se você estiver escrevendo funções que, no verdadeiro espírito do Python, podem tratar com argumentos de qualquer tipo, às vezes é necessário fazer cópias de `x`, independente do que seja `x`. É aí que entra o módulo `copy`. Ele fornece duas funções, `copy` e `deepcopy`. A primeira é exatamente como a operação de fracionamento de seqüência `[:]` ou o método `copy` de dicionários. A segunda é mais sutil e tem a ver com estruturas profundamente aninhadas (daí o termo `deepcopy` – cópia profunda). Pegue o exemplo de cópia de uma lista `(listOne)`, fracionando-a do início ao fim com a construção `[:]`. Essa técnica cria uma nova lista que contém referências para os mesmos objetos contidos na lista original. Se o conteúdo dessa lista original são objetos imutáveis, como números ou strings, a cópia é tão boa quanto uma cópia "verdadeira". Entretanto, suponha que o primeiro elemento em `listOne` seja um dicionário (ou qualquer outro objeto mutável). O primeiro elemento da cópia de `listOne` é uma nova referência para o mesmo dicionário. Assim, se você modificar esse dicionário, a modificação ficará evidente tanto em `listOne` como na cópia de `listOne`. Um exemplo torna isso muito mais claro:

```
>>> import copy
>>> listOne = [{"name": "Willie", "city": "Providence, RI"}, 1, "tomato", 3.0]
>>> listTwo = listOne[:]                # Ou listTwo=copy.copy(listOne)
>>> listThree = copy.deepcopy(listOne)
>>> listOne.append("kid")
>>> listOne[0]["city"] = "San Francisco, CA"
>>> print listOne, listTwo, listThree
[{'name': 'Willie', 'city': 'San Francisco, CA'}, 1, 'tomato', 3.0, 'kid']
[{'name': 'Willie', 'city': 'San Francisco, CA'}, 1, 'tomato', 3.0]
[{'name': 'Willie', 'city': 'Providence, RI'}, 1, 'tomato', 3.0]
```

Como você pode ver, modificar `listOne` diretamente modificou apenas `listOne`. Modificar a primeira entrada da lista referenciada por `listOne` leva a alterações em `listTwo`, mas não em `listThree`; essa é a diferença entre uma cópia rasa `([:])` e uma cópia profunda. As funções do módulo `copy` sabem como copiar todos os tipos internos que podem ser copiados razoavelmente*, incluindo classes e instâncias.

Ordenação

As listas têm um método `sort` que realiza ordenação no local. Às vezes, você quer iterar no conteúdo ordenado de uma lista, sem perturbar o conteúdo dessa lista. Ou então, talvez você queira listar o conteúdo ordenado de uma tupla. Como as tuplas são imutáveis, uma operação

* Alguns objetos não se qualificam como os "que podem ser copiados razoavelmente", como módulos, objetos arquivo e soquetes. Lembre-se de que os objetos arquivo são diferentes dos arquivos em disco, pois são abertos em um ponto específico e possivelmente ainda nem mesmo estão totalmente gravados no disco. Para cópia de arquivos em disco, o módulo `shutil` será apresentado posteriormente neste capítulo.

como `sort`, que a modifica no local, não é permitida. A única solução é fazer uma cópia dos elementos da lista, ordenar a cópia da lista e trabalhar com a cópia ordenada, como em:

```
listCopy = list(myTuple)
listCopy.sort()
for item in listCopy:
    print item                         # Ou o que precisar ser feito
```

Essa solução também é a maneira de tratar com estruturas de dados que não têm nenhuma ordem inerente, como os dicionários. Um dos motivos pelos quais os dicionários são tão rápidos é que a implementação se reserva o direito de mudar a ordem das chaves neles presentes. Entretanto, isso não é um problema real, dado que você pode iterar pelas chaves de um dicionário usando uma cópia intermediária das chaves do dicionário:

```
keys = myDict.keys()                   # Retorna uma lista não ordenada
                                       # das chaves do dicionário.
keys.sort()
for key in keys:                       # Imprime pares chave/valor
    print key, myDict[key]             # ordenados pela chave.
```

Em listas, o método `sort` usa o esquema de comparação padrão do Python. Às vezes, entretanto, esse esquema não é o suficiente e você precisa ordenar de acordo com algum outro procedimento. Por exemplo, ao ordenar uma lista de palavras, a caixa (minúsculas *versus* MAIÚSCULAS) pode não ser significativa. Entretanto, a comparação padrão de strings de texto diz que todas as letras maiúsculas vêm antes de todas as letras minúsculas. Portanto, `'Baby'` é menor do que `'apple'`, mas `'baby'` é maior do que `'apple'`. Para realizar uma ordenação independente da caixa, você precisa definir uma função de comparação que receba dois argumentos e retorne –1, 0 ou 1, dependendo do primeiro argumento ser menor, igual ou maior do que o segundo. Assim, para uma ordenação independente da caixa, você pode usar:

```
>>> def caseIndependentSort(something, other):
...     something, other = something.lower(), other.lower()
...     return cmp(something, other)
...
>>> testList = ['this', 'is', 'A', 'sorted', 'List']
>>> testList.sort()
>>> print testList
['A', 'List', 'is', 'sorted', 'this']
>>> testList.sort(caseIndependentSort)
>>> print testList
['A', 'is', 'List', 'sorted', 'this']
```

Estamos usando a função interna `cmp`, que faz a parte difícil de descobrir que `'a'` vem antes de `'b'`, `'b'` vem antes de `'c'` etc. Nossa função de ordenação simplesmente converte os dois itens para minúsculas e compara as versões em minúsculas. Note também que a conversão para minúscula é local para a função de comparação, de modo que os elementos da lista não são modificados pela ordenação.

Tornando aleatório

E quanto ao fato de tornar uma seqüência, como uma lista de linhas, aleatória? O modo mais fácil de tornar uma seqüência aleatória é chamar a função `shuffle` no módulo `random`, que torna uma seqüência aleatória no local:*

* Outra função útil no módulo `random` é `choice`, que retorna um elemento aleatório da seqüência passada como argumento.

O projetista da classe espera que tais atributos pseudo-privados sejam usados apenas pelos métodos da classe e pelos métodos de qualquer subclasse eventual.

Fazendo novas listas e dicionários

A classe `Stack`, apresentada anteriormente, executa seu trabalho mínimo muito bem. Ela presume uma definição realmente mínima do que uma pilha é; especificamente, algo que suporta apenas duas operações, `push` e `pop`. Seria ótimo usar algumas das características das listas, como a capacidade de iteração sobre todos os elementos, usando a construção `for...in...` Embora pudéssemos continuar no estilo da classe anterior e delegar para o objeto lista "interno", em algum ponto faz mais sentido simplesmente reutilizar a implementação de objetos lista diretamente, por meio de subclasses. Neste caso, você deve derivar uma classe da classe de base `list`. A classe de base `dict` também pode ser usada para criar classes do tipo dicionário.

```
# Subclasse da classe list.
class Stack(list):
    push = list.append
```

Aqui, `Stack` é uma subclasse da classe `list`. A classe `list` implementa o método `pop`, dentre outros. Você não precisa definir seu próprio método `__init__`, pois `list` define um padrão perfeitamente bom. O método `push` é definido apenas dizendo-se que ele é igual ao método `append` de `list`. Agora, podemos fazer coisas do tipo lista, assim como coisas do tipo pilha:

```
>>> thingsToDo = Stack(['write to mom', 'invite friends over', 'wash the kid'])
>>> print thingsToDo                  # Herdada da classe de base list
['write to mom', 'invite friends over', 'wash the kid']
>>> thingsToDo.pop()
'wash the kid'
>>> thingsToDo.push('change the oil')
>>> for chore in thingsToDo:          # Também podemos iterar pelo conteúdo.
...     print chore
...
write to mom
invite friend over
change the oil
```

MANIPULANDO ARQUIVOS E DIRETÓRIOS

Até aqui, tudo bem – sabemos criar objetos, podemos convertê-los entre diferentes tipos de dados e podemos executar vários tipos de operações sobre eles. Na prática, contudo, assim que a pessoa sai do curso de ciência da computação, se depara com tarefas que envolvem manipulação de dados que residem fora do programa e executam processos externos ao Python. É aí que se torna muito útil saber se comunicar com o sistema operacional, explorar o sistema de arquivos, ler e modificar arquivos.

Os módulos os e os.path

O módulo `os` fornece uma interface genérica para o conjunto de ferramentas mais básico do sistema operacional. Diferentes sistemas operacionais possuem comportamentos diferentes. Isso também vale para a interface de programação. Esse fato torna difícil escrever os assim chamados programas "portáveis", que funcionam bem independentemente do sistema operacional. Ter interfaces genéricas, independentes do sistema operacional, ajuda, assim como usar uma

linguagem interpretada como o Python. O conjunto de chamadas específicas que o módulo `os` define depende da plataforma que você utiliza. (Por exemplo, as chamadas relacionadas à permissão só estão disponíveis em plataformas que as suportam, como Unix e Windows.) Contudo, recomenda-se que você sempre use o módulo `os`, em vez das versões do módulo específicas de uma plataforma (chamadas de nomes como `posix`, `nt` e `mac`). A Tabela 27-4 lista algumas das funções do módulo `os` usadas mais freqüentemente. Ao nos referirmos aos arquivos no contexto do módulo `os`, estamos nos referindo a nomes de arquivo e não a objetos arquivo.

Tabela 27-4 Funções mais freqüentemente usadas do módulo os

Nome da função	Comportamento

`getcwd()`	Retorna uma string referindo o diretório de trabalho corrente (`cwd` — Current Working Directory):
	`>>> print os.getcwd()`
	`h:\David\book`

`listdir(path)`	Retorna uma lista de todos os arquivos no diretório especificado:
	`>>> os.listdir(os.getcwd())`
	`['preface.doc', 'part1.doc', 'part2.doc']`

`chown(path, uid, gid)`	Altera o ID do proprietário e o ID do grupo do arquivo especificado
`chmod(path, mode)`	Altera as permissões do arquivo especificado com modo numérico mode (por exemplo, 0644 significa leitura/gravação para o proprietário, leitura para todos os outros)
`rename(src, dest)`	Muda o nome do arquivo `scr` para o nome `dest`
`remove(path)` ou `unlink(path)`	Exclui o arquivo especificado (veja `rmdir()` para remover diretórios)
`rmdir(path)`	Exclui o diretório especificado
`removedirs(path)`	Funciona como `rmdir()`, exceto que, se o diretório-folha for removido com sucesso, os diretórios correspondentes aos segmentos do caminho mais à direita serão cortados.
`mkdir(path[, mode])`	Cria um diretório chamado path com o modo numérico mode (veja `os.chmod`): `>>> os.mkdir('newdir')`
`makedirs(path[, mode])`	Igual a `mkdir()`, mas cria todos os diretórios de nível intermediário necessários para conter o diretório-folha: `>>> os.makedirs('newdir/newsubdir/newsubsubdir')`
`system(command)`	Executa o comando `shell` em um subshell; o valor de retorno é o código de retorno do comando
`symlink(src, dest)`	Cria um vínculo condicional do arquivo `src` para o arquivo `dest`
`link(scr, dest)`	Cria um vínculo incondicional do arquivo `src` para o arquivo `dest`

`stat(path)`	Retorna dados sobre o arquivo, como tamanho, hora da última modificação e posse:
	`>>> os.stat('TODO.txt')`
	`# Isso retorna algo como uma tupla.`
	`(33206, 0L, 3, 1, 0, 0, 1753L,`
	`1042186004,1042186004, 1042175785)`
	`>>> os.stat('TODO.txt').st_size`
	`# Vê apenas o tamanho.`
	`1753L`
	`>>> time.asctime(time.localtime`
	`(os.stat('TODO.txt').st_mtime))`
	`'Fri Jan 10 00:06:44 2003'`

`walk(top, topdown=True,onerror=None)` `(Python 2.3 and later)`	Para cada diretório na árvore cuja raiz está em `top` (including top, mas excluindo `'.'` e `'..'`), gera uma tupla de três elementos: `dirpath, dirnames, filenames`

```
MyStartDir                              # Impresso pelo shell
0                                       # Código de retorno de echo
```

O módulo os também contém um conjunto de strings que definem maneiras portáveis de referir-se às partes da sintaxe de nome de arquivo relacionadas ao diretório, como mostrado na Tabela 27-5.

Tabela 27-5 Atributos de string do módulo os

Nome do atributo	Significado e valores
curdir	Uma string que denota o diretório corrente: '.' no Unix, DOS e Windows; ':' no Mac
pardir	Uma string que denota o diretório pai: '..' no Unix, DOS e Windows; '::' no Mac
sep	O caractere que separa componentes do nome de caminho: '/' no Unix, '\' no DOS e Windows, ':' no Mac
altsep	Um caractere alternativo para sep, quando disponível; configure como None em todos os sistemas, exceto no DOS e no Windows, onde ele é '/'
pathsep	O caractere que separa componentes de caminho: ':' no Unix, ';' no DOS e Windows

Essas strings são usadas pelas funções do módulo os.path, o qual manipula caminhos de arquivo de maneiras portáveis (consulte a Tabela 27-6). Note que o módulo os.path é um atributo do módulo os e não um sub-módulo de um pacote os; ele é importado automaticamente, quando o módulo os é carregado, e (ao contrário dos pacotes) você não precisa importá-lo explicitamente. As saídas dos exemplos da Tabela 27-6 correspondem à execução do código em uma máquina Windows ou DOS. Em outra plataforma, seriam usados os separadores de caminho apropriados. Um conhecimento relevante e útil é o de que a barra normal (/) pode ser usada com segurança no Windows, para indicar travessia de diretório, mesmo que o separador nativo seja a barra invertida (\) – o Python e o Windows fazem ambos a coisa certa com ela.

Tabela 27-6 Funções mais freqüentemente usadas do módulo os.path

Nome da função	Comportamento

split(path) é equivalente à tupla: (dirname(path), basename(path))	Divide o caminho dado em um par composto de uma cabeça e uma cauda; a cabeça é o caminho até o diretório e a cauda é o nome de arquivo:
	`>>> os.path.split("h:/David/book/part2.doc"`
	`('h:/David/book', 'part2.doc')`

splitdrive(p)	Divide um nome de caminho nos especificadores de unidade de disco e caminho:
	`>>> os.path.splitdrive(r"C:\foo\bar.txt")`
	`('C:', '\\foo\\bar.txt')`

splitext(p)	Divide a extensão de um nome de caminho:
	`>>> os.path.splitext(r"C:\foo\bar.txt")`
	`('C:\\foo\\bar', '.txt')`

splitunc(p)	Divide um nome de caminho nos especificadores de ponto de montagem UNC e caminho relativo:
	`>>> os.path.splitunc(r"\\machine\mount\directory \file.txt")`
	`('\\\\machine\\mount', '\\directory\\file.txt')`

join(path, ...)	Une componentes de caminho de forma inteligente:
	`>>> print os.path.join(os.getcwd(),`
	`... os.pardir, 'backup', 'part2.doc')`
	`h:\David\book\..\backup\part2.doc`

exists(path)	Retorna verdadeiro se path corresponde a um caminho existente

Tabela 27-7 Funções do módulo shutil

Nome da função	Comportamento
`copyfile(src, dest)`	Faz uma cópia do arquivo `src` e o chama de dest (cópia binária direta).
`copymode(src, dest)`	Copia informações de modo (permissões) de `src` em `dest`.
`copystat(src, dest)`	Copia todas as informações estatísticas (modo, utime) de `src` em `dest`.
`copy(src, dest)`	Copia dados e informações de modo de `src` em `dest` (não inclui a bifurcação de recurso em Macs).
`copy2(src, dest)`	Copia dados e informações estatísticas de `src` em `dest` (não inclui a bifurcação de recurso em Macs).
`copytree(src, dest, symlinks=0)`	Copia um diretório recursivamente, usando `copy2`. O flag `symlinks` especifica se os vínculos simbólicos na árvore de origem devem resultar em vínculos simbólicos na árvore de destino ou se os arquivos que estão sendo vinculados devem ser copiados. O diretório de destino ainda não deve existir.
`rmtree(path, ignore_errors=0, onerror=None)`	Exclui recusivamente o diretório indicado pelo caminho. Se `ignore_error` for configurado como 0 (o comportamento padrão), os erros serão ignorados. Caso contrário, se `onerror` for configurado, será chamado para tratar do erro; senão, uma exceção será lançada em caso de erro.

Nomes de arquivo e diretórios

Embora a seção anterior tenha listado funções comuns para se trabalhar com arquivos, muitas tarefas exigem mais do que uma chamada de função simples.

Vamos ver um exemplo típico: você tem muitos arquivos, todos os quais contendo um espaço em seus nomes, e gostaria de substituir os espaços por sublinhados. Tudo que você precisa é do atributo `os.curdir` (que retorna uma string específica do sistema operacional que corresponde ao diretório corrente), da função `os.listdir` (que retorna a lista de nomes de arquivo em um diretório especificado) e da função `os.rename`:

```
import os
if len(sys.argv) == 1:                   # Se o nome de arquivo não
                                         # for especificado,
     filenames = os.listdir(os.curdir)   # usa o dir corrente;
else:                                    # caso contrário, usa os
                                         # arquivos especificados
     filenames = sys.argv[1:]            # na linha de comando.
for filename in filenames:
     if ' ' in filename:
         newfilename = filename.replace(' ', '_')
         print "Renaming", filename, "to", newfilename, "..."
         os.rename(filename, newfilename)
```

Esse programa funciona bem, mas revela certa queda para o Unix. Isto é, se você o chamar com curingas, como em:

```
python despacify.py *.txt
```

verá que, em máquinas Unix, ele altera o nome de todos os arquivos com nomes que contêm espaços e que terminam com `.txt`. Entretanto, em um shell estilo DOS, isso não funcionaria, pois o shell normalmente usado no DOS e no Windows não converte **.txt* na lista de nomes

```
# Cria arquivo de saída final
outputFile = open('output.txt', 'w')
second_process(input = tempFile, output = outputFile)
```

O uso de `tempfile.TemporaryFile()` funciona bem nos casos onde as etapas intermediárias manipulam objetos arquivo. Uma de suas características interessantes é que, quando o objeto arquivo é excluído, ele exclui automaticamente o arquivo que criou no disco, fazendo a limpeza após ele mesmo. Entretanto, um uso importante dos arquivos temporários é em conjunto com a chamada de `os.system`, o que significa usar um shell, usando com isso nomes de arquivo e não objetos arquivo. Por exemplo, vamos ver um programa que cria cartas padronizadas e as envia para uma lista de endereços de email (apenas no Unix):

```
formletter = """Dear %s, \nI'm writing to you to suggest that ..."""   # etc.
myDatabase = [('Michael Jackson', 'michael@neverland.odd'),
               ('Bill Gates', 'bill@microsoft.com'),
               ('Bob', 'bob@subgenius.org')]
for name, email in myDatabase:
    specificLetter = formletter % name
    tempfilename = tempfile.mktemp()
    tempfile = open(tempfilename, 'w')
    tempfile.write(specificLetter)
    tempfile.close()
    os.system('/usr/bin/mail %(email)s -s "Urgent!" < %(tempfilename)s' % vars())
    os.remove(tempfilename)
```

A primeira linha no loop `for` retorna uma versão personalizada da carta padronizada com base no nome recebido. Então, esse texto é gravado em um arquivo temporário que é enviado para o endereço de email apropriado, usando a chamada de `os.system`. Finalmente, para fazer a limpeza, o arquivo temporário é removido.

`vars()` é uma função interna que retorna um dicionário correspondente às variáveis definidas no espaço de nome local corrente. As chaves do dicionário são os nomes das variáveis e os valores do dicionário são os valores das variáveis. A função `vars()` é muito útil para explorar espaços de nome. Ela também pode ser chamada com um objeto como argumento (como um módulo, uma classe ou uma instância) e retornará o espaço de nome desse objeto. Duas outras funções internas, `locals()` e `globals()`, retornam os espaços de nome local e global, respectivamente. Em todos os três casos, modificar os dicionários retornados não garante nenhum efeito no espaço de nome em questão; portanto, considere-os como somente para leitura e você não será surpreendido. Você pode ver que a chamada de `vars()` cria um dicionário que é usado pelo mecanismo de interpolação de string. Assim, é importante que os nomes dentro dos bits `%(...)s` na string correspondam aos nomes de variável no programa.

Modificando entradas e saídas

O atributo `argv` do módulo `sys` contém um conjunto de entradas para o programa corrente – os argumentos de linha de comando; mais precisamente, uma lista das palavras inseridas na linha de comando, excluindo a referência para o próprio Python, se ela existir. Em outras palavras, se você digitar no shell:

```
csh> python run.py a x=3 foo
```

então, quando *run.py* começar, o valor do atributo `sys.argv` será `['run.py', 'a', 'x=3', 'foo']`. O atributo `sys.argv` é mutável (afinal, trata-se apenas de uma lista). A utilização comum envolve a iteração sobre os argumentos do programa em Python (ou seja, `sys.`

`argv[1:]`); o fracionamento do índice 1 até o final fornece todos os argumentos do programa em si, mas não inclui o nome do programa (módulo) armazenado em `sys.argv[0]`. Existem dois módulos que ajudam a processar opções de linha de comando. O primeiro, um módulo mais antigo, chamado `getopt`, é substituído no Python 2.3 por um módulo semelhante, porém mais poderoso, chamado `optparse`. Consulte a referência da biblioteca para ver mais detalhes sobre como utilizá-los.

Os programadores experientes sabem que existem outras entradas para um programa, especialmente o fluxo de entrada padrão, com congêneres para saída e mensagens de erro. O Python permite que o programador as acesse e modifique por meio de três atributos de arquivo no módulo `sys: sys.stdin, sys.stdout e sys.sdterr`. Geralmente, a entrada padrão é associada, pelo sistema operacional, ao teclado do usuário; a saída padrão e o erro padrão normalmente são associados ao console. No Python, a instrução `print` gera saída na saída padrão (`sys.stdout`), enquanto mensagens de erro, como as exceções, vão para o fluxo de erro padrão (`sys.stdterr`). O Python permite que você modifique isso dinamicamente: você pode redirecionar a saída de um programa em Python para um arquivo, simplesmente atribuindo a `sys.stdout`:

```
sys.stdout = open('log.out', 'w')
```

Após essa linha, toda saída será gravada no arquivo `log.out`, em vez de aparecer no console. Note que, se você não salvá-la primeiro, a referência para o fluxo de saída padrão "original" será perdida. Geralmente é uma boa idéia salvar uma referência antes de realocar qualquer um dos fluxos padrão, como em:

```
old_stdout = sys.stdout
sys.stdout = open('log.out', 'w')
```

Usando E/S padrão para processar arquivos

Por que ter um fluxo de entrada padrão? Afinal, não é tão difícil digitar `open('input.txt')` no programa. O principal argumento para ler e gravar com fluxos padrão é que você pode encadear programas de modo que a saída padrão de um se torne a entrada padrão do seguinte, sem nenhum arquivo usado na transferência. Essa facilidade, conhecida como *canalização*, é o centro da filosofia Unix. Usar E/S padrão dessa maneira significa que você pode escrever um programa para executar uma tarefa específica uma vez e, então, usá-lo para processar arquivos ou resultados intermediários de outros programas a qualquer momento no futuro.

Como exemplo, um programa simples que conta o número de linhas em um arquivo, poderia ser escrito como:

```
import sys
data = sys.stdin.readlines()
print "Counted", len(data), "lines."
```

No Unix, você poderia testá-lo com algo como:

```
% cat countlines.py | python countlines.py
Counted 3 lines.
```

No Windows ou no DOS, você escreveria:

```
C:\> type countlines.py | python countlines.py
Counted 3 lines.
```

Você pode obter cada linha de um arquivo simplesmente iterando em um objeto arquivo. Isso é muito útil na implementação de operações de filtro simples. Aqui estão alguns exemplos de tais operações de filtro.

Localizando todas as linhas que começam com

```
# Mostra linhas de comentário (linhas que começam com #, como esta).
import sys
for line in sys.stdin:
    if line[0] == '#':
        print line,
```

Note que uma vírgula final foi adicionada após a instrução `print`, para indicar que a operação de impressão não deve adicionar um caractere de nova linha, o que resultaria em uma saída com espaço duplo, pois a string da linha já inclui um caractere de nova linha como seu último caractere.

Os dois últimos programas podem ser combinados facilmente, usando-se pipes para combinar seu poder. Para contar o número de linhas de comentário em *commentfinder.py*:

```
C:> types commentfinder.py | python commenfinder.py | python countlines.py
Counted 1 lines.
```

Algumas outras tarefas de filtragem que recebem da entrada padrão e gravam na saída padrão aparecem a seguir.

Extraindo a quarta coluna de um arquivo (onde as colunas são definidas por espaços em branco)

```
import sys
for line in sys.stdin:
    words = line.split()
    if len(words) >= 4:
        print words[3]
```

Examinamos o comprimento das palavras para descobrir se realmente existem pelo menos quatro palavras. As duas últimas linhas também poderiam ser substituídas pela instrução `try/except`, o que é muito comum no Python:

```
try:
    print words[3]
except IndexError:                # Não há palavras suficientes.
    pass
```

Extraindo a quarta coluna de um arquivo, onde as colunas são separadas por dois-pontos, e transformando-a em minúscula

```
import sys, string
for line in sys.stdin:
    words = line.split(':')
    if len(words) >= 4:
        print words[3].lower()
```

Se a iteração por todas as linhas não é o que você deseja, basta usar os métodos `readlines()` ou `read()` de objetos arquivo.

Imprimindo as 10 primeiras linhas, as 10 últimas linhas e cada outra linha

```
import sys
lines = sys.stdin.readlines()
sys.stdout.writelines(lines[:10])          # As 10 primeiras linhas
sys.stdout.writelines(lines[-10:])         # As 10 últimas linhas
for lineIndex in range(0, len(lines), 2):  # Obtém 0, 2, 4, ...
    sys.stdout.write(lines[lineIndex])     # Obtém a linha indexada.
```

Contando o número de vezes que a palavra "Python" ocorre em um arquivo

```
text = open(fname).read()
print text.count('Python')
```

Transformando uma lista de colunas em uma lista de linhas

Neste exemplo mais complicado, a tarefa é transpor um arquivo. Imagine que você tenha um arquivo como o seguinte:

```
Name:     Willie   Mark    Guido    Mary     Rachel   Ahmed
Level:      5        4       3        1        6        4
Tag#:     1234     4451    5515     5124     1881     5132
```

E queira que, em vez disso, ele seja como segue:

```
Name:     Level:   Tag#:
Willie    5        1234
Mark      4        4451
...
```

Você poderia usar um código como este:

```
import sys
lines = sys.stdin.readlines()
wordlist = [line.split() for line in lines]
for row in zip(*wordlists):
    print '\t'.join(row)
```

É claro que você poderia usar técnicas de programação muito mais defensivas para tratar da possibilidade de nem todas as linhas conterem o mesmo número de palavras, de estar faltando dados etc. Essas técnicas são específicas da tarefa e ficam como exercício para o leitor.

Escolhendo tamanhos de trecho

Todos os exemplos anteriores presumem que você pode ler o arquivo inteiro de uma vez. Em alguns casos, contudo, isso não é possível; por exemplo, ao processar arquivos realmente grandes em computadores com pouca memória ou ao tratar com arquivos em que anexações estão constantemente sendo feitas (como os arquivos de log). Nesses casos, você pode usar uma combinação de while/readline, onde parte do arquivo é lida um bit por vez, até que se chegue ao final do arquivo. Ao tratar com arquivos que não são orientados a linhas, você deve ler o arquivo um caractere por vez:

```
# Lê caractere por caractere.
while 1:
    next = sys.stdin.read(1)         # Lê uma string de um caractere
    if not next:                     # ou uma string vazia em EOF.
       break
    # Processa o próximo ('next') caractere.
```

Note que o método `read()` nos objetos arquivo retorna uma string vazia no final do arquivo, o que faz sair do loop `while`. Entretanto, mais freqüentemente, os arquivos com que você vai lidar consistem em dados baseados em linhas e são processados uma linha por vez:

```
# Lê linha por linha.
while 1:
    next = sys.stdin.readline(1)       # Lê uma string de uma linha
    if not next:                       # ou uma string vazia em EOF.
        break
# Processa a próxima ('next') linha.
```

Fazendo algo com um conjunto de arquivos especificado na linha de comando

Poder ler `stdin` é um recurso excelente; ele é a base do conjunto de ferramentas do Unix. Entretanto, uma única entrada nem sempre é suficiente: muitas tarefas precisam ser executadas sobre conjuntos de arquivos. Normalmente, isso é resolvido fazendo-se com que o programa em Python analise a lista de argumentos enviados para o script como opções de linha de comando. Por exemplo, se você digitar:

```
% python myScript.py input1.txt input2.txt input3.txt output.txt
```

poderá pensar que *myScript.py* deseja fazer algo com os três primeiros arquivos de entrada e gravar um novo arquivo, chamado *output.txt*. Vamos ver como poderia ser o início de um programa assim:

```
import sys
inputfilenames, outputfilename = sys.argv[1:-1], sys.argv[-1]
for inputfilename in inputfilenames:
    inputfile = open(inputfilename, "r")
    do_something_with_input(inputfile)
inputfile.close()
outputfile = open(outputfilename, "w")
write_results(outputfile)
outputfile.close()
```

A segunda linha extrai partes do atributo `argv` do módulo `sys`. Lembre-se de que essa é uma lista de palavras na linha de comando que chamou o programa corrente. Ela começa com o nome do script. Assim, no exemplo anterior, o valor de `sys.argv` é:

```
['myScript.py', 'input1.txt', 'input2.txt', 'input3.txt', 'output.txt'].
```

O script presume que a linha de comando consiste em um ou mais arquivos de entrada e um arquivo de saída. Então, o fracionamento dos nomes de arquivo de entrada começa em 1 (para pular o nome do script, que não é uma entrada para o script, na maioria dos casos) e pára antes da última palavra na linha de comando, que é o nome do arquivo de saída. O restante do script deve ser muito fácil de entender (mas não funcionará até que você forneça as funções `do_something_with_input()` e `write_results()`).

Note que o script anterior não lê realmente os dados dos arquivos, mas passa o objeto arquivo para uma função, para fazer o trabalho em si. Uma versão genérica de `do_something_with_input()` é:

```
def do_something_with_input(inputfile):
    for line in inputfile:
    process(line)
```

```
version_number, num_bytes = .unpack('fl', data[start:stop])
start, stop = stop, start = struct.calcsize('B'*num_bytes)
bytes = struct.unpack('B'*num_bytes, data[start:stop])
```

`'f'` é uma string de formato para um número de ponto flutuante (um `float` da linguagem C, para sermos precisos). `'l'` significa inteiro longo e `'B'` é uma string de formato para caractere sem sinal. As strings de formato de `unpack` disponíveis estão listadas na Tabela 27-8. Consulte o manual de referência da biblioteca para ver detalhes da utilização.

Tabela 27-8 Códigos de formato comuns usados pelo módulo struct

Formato	Tipo C	Python
x	pad byte	Nenhum valor
c	char	String de comprimento 1
b	signed char	Inteiro
B	unsigned char	Inteiro
h	short	Inteiro
H	unsigned short	Inteiro
i	int	Inteiro
I	unsigned int	Inteiro
l	long	Inteiro
L	unsigned long	Inteiro
f	float	Ponto flutuante
d	double	Ponto flutuante
s	char[]	String
p	char[]	String
P	void*	Inteiro

Neste ponto, bytes é uma tupla de `num_bytes` inteiros do Python. Se soubéssemos que, na verdade, os dados estão armazenando caracteres, poderíamos usar `chars = map(chr bytes)`. Para sermos eficientes, poderíamos alterar a última instrução `unpack` para usar `'c'`, em vez de `'B'`, que faria a conversão e retornaria uma tupla de `num_bytes` strings de um caractere. Mais eficientemente ainda, poderíamos usar uma string de formato que especificasse uma string de caracteres de comprimento determinado, como:

```
chars = struc.unpack(str(num_bytes)+'s', data[start:stop])
```

A operação de empacotamento `(struct.pack)` é exatamente o inverso; em vez de pegar uma string de formato, uma string de dados e retornar uma tupla de valores desempacotados, ela pega uma string de formato, um número variável de argumentos e empacota esses argumentos usando essa string de formato em uma nova string empacotada.

Note que o módulo `struct` pode processar dados escritos com qualquer um dos tipos de ordenação de byte*, permitindo assim que você escreva código de manipulação de arquivo binário independente de plataforma. Para arquivos grandes, considere também o uso do módulo `array`.

* A ordem na qual os computadores listam palavras de vários bytes depende do chip usado (tanto mais para os padrões). Os sistemas Intel e DEC usam a ordem little-endian, enquanto os sistemas Motorola e Sun usam ordem big-endian. As transmissões em rede também usam a ordem big-endian, de modo que o módulo `struct` é útil para fazer E/S de rede em PCs.

MÓDULOS RELACIONADOS À INTERNET

O Python é usado em uma ampla variedade de tarefas relacionadas à Internet; desde fazer servidores da Web navegarem na Web, até "cavoucar" sites da Web em busca de dados. Esta seção descreve sucintamente os módulos usados mais freqüentemente para essas tarefas que acompanham a base do Python. Para exemplos mais detalhados do seu uso, recomendamos o livro *Standard Python Library*, de Lundh, e o livro *Python Cookbook*, de Martelli e Ascher (O'Reilly). Existem muitos complementos de outros fornecedores que valem a pena conhecer, antes de embarcar em um projeto significativo relacionado à Web ou à Internet.

A interface de gateway comum: o módulo cgi

Os programas em Python freqüentemente processam formulários de páginas da Web. Para facilitar essa tarefa, a distribuição de Python padrão inclui um módulo chamado `cgi`. O Capítulo 28 contém um exemplo de script em Python que usa a CGI; portanto, não a abordaremos com mais detalhes aqui.

Manipulando URLs: os módulos urllib e urlparse

Os localizadores de recurso universais são strings como *http://www.python.org* que, agora, são onipresentes. Três módulos – `rullib`, `urllib2` e `urlparse` – fornecem ferramentas para processar URLs.

O módulo `urllib` define algumas funções para escrever programas que devem ser usuários ativos da Web (robôs, agentes etc.). Elas estão listadas na Tabela 27-9.

Tabela 27-9 Funções do módulo urllib

Nome da função	Comportamento

`urlopen(url [, data])`	Abre (para leitura) um objeto de rede denotado por uma URL; ela também pode abrir arquivos locais:
	`>>> page = urlopen('http://www.python.org')`
	`>>> page.readline()`
	`'<HTML>\012'`
	`>>> page.readline()`
	`'<!-- THIS PAGE IS AUTOMATICALLY GENERATED.DO NOT EDIT.`
	`'--> \012'`

`urlretrieve(url [, filename][, hook])`	Copia um objeto de rede denotado por uma URL em um arquivo local (usa uma cache): `>>> urllib.urlretrieve('http://www.python.org/', 'wwwpython.html')`
`urlcleanup()`	Limpa a cache usada por `urlretrieve`.

`quote(string[, safe])`	Substitui caracteres especiais na string usando o escape `%xx`. O parâmetro opcional safe especifica caracteres adicionais que não devem ser colocados entre aspas; seu valor padrão é:
	`>>> quote('this & that @ home')`
	`'this%20%26%20that%20%40%20home'`

`quote_plus(string[, safe])`	Igual a quote`()`, mas também substitui espaços por sinais de adição.

`unquote(string)`	Substitui escapes `%xx` por seus equivalentes de um caractere:
	`>>> unquote('this%20%26%20that%20%40%20home')`
	`'this & that @ home'`

Tabela 27-9 Funções do módulo urllib (continuação)

Nome da função	Comportamento

`urlencode(dict)`	Converte um dicionário em uma string codificada como URL, conveniente para passar para urlopen() como o argumento de dados opcional:
	`>>> locals()`
	`{'urllib': <module 'urllib'>, '__doc__': None, 'x': 3, '__name__': '__main__', '__builtins__': <module '__builtin__'>}`
	`>>> urllib.urlencode(locals())`
	`'urllib=%3cmodule+%27urllib%27%3e&__doc__=None&x=3&__name__=__main__&__builtins__=%3cmodule+%27__builtin__%27%3e'`

O módulo `urllib2` focaliza as tarefas de abertura de URLs que o módulo mais simples `urllib` não sabe como tratar e fornece um modelo extensível para novos tipos de URLs e protocolos. É isso que você deve usar se quiser lidar com senhas, autenticação digest*, proxys, URLs HTTPS e outras URLs interessantes.

O módulo `urlparse` define algumas funções que simplificam a separação de URLs e a composição de novas URLs. Elas estão listadas na Tabela 27-10.

Tabela 27-10 Funções do módulo urlparse

Nome da função	Comportamento

`urlparse(urlstring[,default_scheme[,allow fragments]])`	Passa uma URL para seis componentes, retornando uma tupla de seis elementos (esquema de endereçamento, localização na rede, caminho, parâmetros, consulta, identificador de fragmento):
	`>>> urlparse('http://www.python.org/' FAQ.html')`
	`('http', 'www.python.org', '/FAQ.html', '', '', '')`

`urlunparse(tuple)`	Constrói uma string de URL a partir de uma tupla, conforme retornado por `urlparse()`

`urljoin(base, url[,allow fragments])`	Constrói uma URL completa (absoluta), combinando uma URL de base `(base)` com uma URL relativa `(url)`:
	`>>> urljoin('http://www.python.org', 'doc/lib')`
	`'http://www.python.org/doc/lib'`

Protocolos específicos de Internet

Os protocolos mais comumente usados, construídos sobre TCP/IP, são suportados com módulos cujos nomes os revelam. O módulo `telnetlib` permite que você atue como um cliente de Telnet. O módulo `httplib` permite que você se comunique com servidores da Web com o protocolo HTTP. O módulo `ftplib` serve para transferir arquivos usando o protocolo FTP. O módulo `gopherlib` serve para navegar em servidores Gopher (agora muito raros). Nos domínios do correio e notícias, você pode usar os módulos `poplib` e `imaplib` para ler arquivos de correio em servidores POP3 e IMAP, respectivamente, o módulo `smtplib` para enviar correspondência e o módulo `nntplib` para ler e postar notícias na Usenet a partir de servidores NNTP.

Também existem módulos que podem construir servidores de Internet; especificamente, um servidor de IP genérico baseado em soquete `(SocketServer)`, um servidor da Web simples

* Nota de R.T.: Autenticação de resenha. Uma forma de autenticação para navegadores na internet.

(`SimpleHTTPServer`), um servidor de HTTP compatível com CGI (`CGIHTTPServer`) e um módulo para construir serviços de manipulação de soquetes assíncronos (`asyncore`).

Atualmente, o suporte para serviços da Web consiste em uma biblioteca básica para processar chamadas XML-RPC no lado do cliente (`xmlrpclib`), assim como uma implementação de servidor XML-RPC simples (`SimpleXMLRPCServer`). É provável que vá ser adicionado suporte para SOAP, quando o padrão SOAP se tornar mais estável.

Processando dados de Internet

Quando você usa um protocolo de Internet para obter arquivos da Internet (ou antes de enviá-los na Internet), freqüentemente precisa processar esses arquivos. Eles aparecem em muitos formatos diferentes. A Tabela 27-11 lista cada módulo da biblioteca padrão que processa um tipo específico de formato de arquivo relacionado à Internet (existem outros para processamento de formato de som e imagem; consulte o manual de referência da biblioteca).

Tabela 27-11 Módulos dedicados ao processamento de arquivos de Internet

Nome do módulo	Formato de arquivo
`sgmllib`	Um analisador simples para arquivos SGML.
`htmllib`	Um analisador para documentos HTML.
`formatter`	Formatador de saída genérico e interface de dispositivo.
`rfc822`	Analisa cabeçalhos de correio RFC-822 (isto é, "Subject: hi there!").
`mimetools`	Ferramentas para analisar miolos de mensagem estilo MIME (também conhecidos como anexos de arquivo).
`multifile`	Suporte para ler arquivos que contêm partes distintas.
`binhex`	Codifica e decodifica arquivos no formato binhex4.
`uu`	Codifica e decodifica arquivos no formato uuencode.
`binascii`	Converte entre binário e várias representações codificadas em ASCII.
`xdrlib`	Codifica e decodifica dados XDR.
`mailcap`	Manipulação de arquivo mailcap.
`mimetypes`	Mapeamento de extensões de nome de arquivo para tipos MIME.
`base64`	Codifica e decodifica codificação MIME base64.
`quopri`	Codifica e decodifica codificação de imprimíveis entre aspas MIME.
`mailbox`	Lê vários formatos de caixa de correio.
`mimify`	Converte mensagens de correio para (e do) formato MIME.
`mail`	Um pacote para analisar, manipular e gerar mensagens de email.

Processamento de XML

O Python vem com um rico conjunto de ferramentas de processamento de XML. Elas incluem analisadores, interfaces DOM, interfaces SAX e muito mais, como se vê na Tabela 27-12.

Tabela 27-12 Alguns dos módulos para XML da distribuição básica

Nome do módulo	Descrição
`xml.parsers.expat`	Uma interface para o analisador de XML sem validação Expat
`xml.dom`	API DOM (Document Object Model) para o Python
`xml.dom.minidom`	Implementação leve do DOM
`xml.dom.pulldom`	Suporte para construção de árvores DOM parciais a partir de eventos SAX

```
>>> code = "x = 'Something'"
>>> x = "Nothing"                    # Configura o valor de x
>>> exec code                        # Modifica o valor de x!
>>> print x
'Something'
```

A instrução `exec` pode receber argumentos opcionais. Se um único argumento de dicionário for fornecido (após a então obrigatória palavra `in`), ele será usado como espaços de nome local e global para a execução do código especificado. Se forem fornecidos dois argumentos de dicionário, eles serão usados como espaços de nome global e local, respectivamente. Se os dois argumentos forem omitidos, como no exemplo anterior, os espaços de nome global e local correntes serão usados.

Quando a instrução `exec` é chamada, o Python precisa analisar o código que está sendo executado. Isso pode ser um processo dispendioso em termos de computação, especialmente se um código grande precisar ser executado milhares de vezes. Se esse for o caso, vale a pena compilar o código primeiro (uma vez) e executá-lo quantas vezes for necessário. A função `compile` recebe uma string contendo o código Python e retorna um objeto código compilado, o qual pode então ser processado eficientemente pela instrução `exec`.

A função `compile` recebe três argumentos. O primeiro é a string de código. O segundo é o nome de arquivo correspondente ao arquivo-fonte Python (ou `'<string>'`, se não foi lido de um arquivo); ele é usado no rastreamento `(traceback)`, no caso de uma exceção ser gerada ao se executar o código. O terceiro argumento é `'single'`, `'exec'` ou `'eval'`, dependendo do código ser uma única instrução cujo resultado seria impresso (exatamente como no interpretador interativo), um conjunto de instruções ou uma expressão (criando um objeto código compilado para uso pela função `eval`).

Uma função relacionada é a função interna `execfile`. Seu primeiro argumento deve ser o nome de arquivo de um script em Python, em vez de um objeto arquivo ou string (lembre-se de que os objetos arquivo são as coisas que a função interna `open` retorna, quando é passado um nome de arquivo). Assim, se você quiser que seu script em Python comece executando seus argumentos como scripts do Python, pode escrever algo como o seguinte:

```
import sys
for argument in sys.argv[1:]:     # Vamos pular a nós mesmos;
                                  # senão, isto será para sempre!
    execfile(argument)            # Tanto faz.
```

Dois alertas são justificados com relação a `execfile`. Primeiro, é lógico, mas às vezes surpreendente, que `execfile` executa em seu escopo local por padrão – assim, chamar `execfile` dentro de uma função, freqüentemente terá efeitos muito mais localizados do que os usuários esperam. Segundo, `execfile` quase nunca é a resposta certa – se você estiver escrevendo o código que está sendo executado, deve colocá-lo em um módulo e importá-lo. O comportamento que você obterá será muito mais previsível, seguro e fácil de manter – é muito fácil o código de `execfile()` arruinar o módulo, chamando `execfile`.

Mais duas funções podem executar código Python. A primeira é `eval`, que recebe uma string de código (e o agora esperado par opcional de dicionários) ou um objeto código compilado e retorna a avaliação dessa expressão. Por exemplo:

```
>>> word = 'xo'
>>> z = eval("word*10")
>>> print z
'xoxoxoxoxoxoxoxoxoxo'
```

Depurando com pdb

A primeira tarefa é, não surpreendentemente, a depuração. A distribuição padrão do Python inclui um depurador chamado pdb. Usar o pdb é muito simples. Você importa o módulo pdb e chama seu método run com o código Python que o depurador deve executar. Por exemplo, se você estiver depurando o programa em *spam.py*, faça o seguinte:

```
>>> import spam                     # Importa o módulo que queremos depurar.
>>> import pdb                      # Importa pdb.
>>> pdb.run('instance = spam.Spam()')     # Inicia o pdb com uma
                                          # instrução para executar.
> <string>(0)?()
(Pdb) break spam.Spam.__init__        # Podemos configurar pontos de
                                      # verificação.
(Pdb) next
> <string>(1)?()
(Pdb) n                           # 'n' é a abreviatura de 'next'.
> spam.py(3)__init__()
-> def __init__(self):
(Pdb) n
> spam.py(4)__init__()
-> Spam.numInstances = Spam.numInstances + 1
(Pdb) list                        # Mostra a listagem do código-fonte.
  1   class Spam:
  2        numInstances = 0
  3 B      def __init__(self):    # Observe o B de Breakpoint (ponto de
                                  # verificação).
  4 ->         Spam.numInstances = Spam.numInstances + 1  # Onde estamos
  5        def printNumInstances(self):
  6            print "Number of instances created: ", Spam.numInstances
  7
[EOF]
(Pdb) where                           # Mostra a pilha de chamada.
<string>(1)?()
> spam.py(4)__init__()
-> Spam.numInstances = Spam.numInstances + 1
(Pdb) Spam.numInstances = 10     # Note que podemos modificar variáveis
(Pdb) print Spam.numInstances    # enquanto o programa está sendo depurado.
10
(Pdb) continue                   # Isto continua até o próximo ponto
--Return--                       # de verificação, mas não há nenhum; portanto,
-> <string>(1)?()->None          # terminamos.
(Pdb) c                          # Isso acaba terminando o Pdb.
<spam.Spam instance at 80ee60>   # Esta é a instância retornada.
>>> instance.numInstances        # Note que a alteração em numInstance
11                               # foi feita antes da op de incremento.
```

Conforme a sessão anterior mostra, com o pdb você pode listar o código corrente que está sendo depurado (com uma seta apontando para a linha que está para ser executada), examinar variáveis, modificar variáveis e configurar pontos de verificação. O Capítulo 9 da Library Reference aborda o depurador com detalhes. Existem muitos depuradores alternativos, desde um no IDLE até os mais completos, que você encontrará em IDEs comerciais para o Python.

E a execução desse programa produz:

```
csh> python timings.py
Running lots_of_appends 100 times took 7.891 seconds
Running one_multiply 100 times took 0.120 seconds
```

Conforme você pode ter suspeitado, uma única multiplicação de lista é muito mais rápida do que muitas anexações. Note que nas cronometragens nem sempre é uma boa idéia comparar muitas execuções de funções, em vez de apenas uma. Caso contrário, as cronometragens provavelmente serão fortemente influenciadas por coisas que nada têm a ver com o algoritmo, como tráfego de rede no computador ou eventos de GUI. O Python 2.3 introduz um novo módulo, chamado `timeit`, que fornece uma maneira muito simples de fazer corretamente a cronometragem.

E se você tivesse escrito um programa complexo e ele estivesse executando mais lentamente do que o desejado, mas não tivesse certeza de onde está o problema? Nesse caso, o que você precisa fazer é traçar o perfil do programa: determinar quais partes dele consomem tempo e verificar se elas podem ser otimizadas, ou se a estrutura do programa pode ser modificada para eliminar os gargalos. A distribuição do Python inclui as ferramentas certas para isso: o módulo `profile`, documentado na Library Reference, e outro módulo, `hotspot`, que infelizmente não estava bem documentado quando este livro estava sendo produzido. Supondo que você queira traçar o perfil de determinada função no espaço de nome atual, faça o seguinte:

```
>>> from timings import *
>>> from makezeros import *
>>> import profile
>>> profile.run('do_timing(100, (lots_of_appends, one_multiply))')
Running lots_of_appends 100 times took 8.773 seconds
Running one_multiply 100 times took 0.090 seconds
203 function calls in 8.823 CPU seconds
Ordered by: standard name
ncalls  tottime  percal  cumtime   percall filename:lineno(function)
     100    8.574   0.086    8.574    0.086    makezeros.py:1(lots_of_appends)
     100    0.101   0.001    0.101    0.001    makezeros.py:6(one_multiply)
       1    0.001   0.001    8.823    8.823    profile:0(do timing(100,
(lots_of_appends, one_multiply)))
       0    0.000            0.000             profile:0(profiler)
       1    0.000   0.000    8.821    8.821    python:0(194.C.2)
       1    0.147   0.147    8.821    8.821    timings.py:2(do_timing)
```

Conforme você pode ver, isso fornece uma listagem bastante complicada que inclui coisas como tempo por chamada gasto em cada função e o número de chamadas feitas para cada função. Em programas complexos, o traçador de perfil pode ajudar a encontrar ineficiências surpreendentes. A otimização de programas em Python está fora dos objetivos deste livro; entretanto, se você estiver interessado, consulte o newsgroup do Python: periodicamente, um usuário pede ajuda para tornar um programa mais rápido e começa um debate espontâneo, com conselhos interessantes dados por usuários especialistas.

EXERCÍCIOS

Este capítulo está repleto de programas, os quais o estimulamos a digitar e experimentar. Entretanto, aqui estão mais alguns exercícios desafiadores:

1. *Evitando expressões regulares.* Escreva um programa que atenda aos mesmos requisitos de *pepper.py*, mas não use expressões regulares para fazer o trabalho. Este exercício é bem difícil, mas útil na construção de lógica de programa.

2. *Encerrando um arquivo de texto com uma classe*. Escreva uma classe que receba um nome de arquivo e leia os dados do arquivo correspondente como texto. Faça isso de modo que essa classe tenha três atributos: `paragraph`, `line`, `word`, cada um dos quais recebendo um argumento inteiro, de modo que se *mywrapper* for uma instância dessa classe, imprimir `mywrapper.paragraph(0)` imprime o primeiro parágrafo do arquivo, `mywrapper.line(-2)` imprime a penúltima linha do arquivo e `mywrapper.word(3)` imprime a quarta palavra do arquivo.
3. *Descrevendo um diretório*. Escreva uma função que receba um nome de diretório e descreva o conteúdo desse diretório, recursivamente (em outras palavras, para cada arquivo, imprima o nome e o tamanho, e desça para quaisquer diretórios eventuais).
4. *Modificando o prompt*. Modifique seu interpretador de modo que o prompt, em vez da string `>>>`, seja uma string descrevendo o diretório corrente e a contagem do número de linhas inseridas na sessão de Python corrente. Duas dicas: as variáveis de prompt (por exemplo, `sys.ps1`) não precisam ser strings, mas podem ser qualquer objeto; a impressão de uma instância pode ter efeitos colaterais e é feita chamando-se o método `__repr__` da instância.
5. *Escrevendo um shell*. Usando a classe `Cmd` do módulo `cmd` e as funções descritas neste capítulo para manipular arquivos e diretórios, escreva um pequeno shell que aceite os comandos padrão do Unix (ou comandos do DOS): `ls (dir)` para listar o diretório corrente, `cd` para mudar de diretório, `mv` (ou `ren`) para mover/renomear um arquivo e `cp (copy)` para copiar um arquivo.
6. *Redirecionando stdout*. Modifique o script *mygrep.py* para produzir a saída do último arquivo especificado na linha de comando, em vez de usar o console.

28 Modelos

Até aqui, todos os exemplos deste livro foram muito pequenos e podem parecer brincadeira se comparados com aplicativos do mundo real. Este capítulo mostra alguns dos modelos disponíveis para programadores de Python que desejam construir tais aplicativos em alguns domínios específicos. Um *modelo* pode ser considerado como um conjunto de classes de um domínio específico e padrões esperados de interações entre essas classes. Mencionamos apenas três aqui: o modelo COM para interação com o Component Object Model da Microsoft, a interface gráfica com o usuário (GUI) Tkinter e o kit de ferramentas de GUI Java Swing.

Ilustraremos o poder dos modelos usando um cenário hipotético de um site na Web de uma pequena empresa e a necessidade de coletar, manter e responder às solicitações dos clientes a respeito do produto, por meio de um formulário da Web. Descreveremos três programas nesse cenário. O primeiro é um formulário de entrada de dados baseado na Web que pede para o usuário digitar algumas informações em seu navegador e depois salva essas informações no disco. O segundo programa usa os mesmos dados e utiliza o Microsoft Word automaticamente, para imprimir uma carta padronizada personalizada, baseada nessas informações. O último exemplo é um navegador simples para os dados salvos, construído com o módulo Tkinter, que utiliza a GUI Tk, um kit de ferramentas poderoso e portável para gerenciar janelas, botões, menus etc. Esperamos que esses exemplos façam com que você perceba como esses tipos de kits de ferramentas, quando combinados com o poder de desenvolvimento rápido do Python, podem realmente permitir a construção rápida de aplicativos reais. Cada programa complementa o anterior, de modo que recomendamos veementemente que você leia cada um deles, mesmo que não queira que estejam prontos e funcionando em seu computador.

A última seção deste capítulo aborda o Jython, a implementação Java do Python. O capítulo termina com um programa em Jython de tamanho médio, que permite aos usuários manipularem funções matemáticas graficamente, usando o kit de ferramentas Swing.

UM SISTEMA DE RECLAMAÇÃO AUTOMATIZADO

O cenário deste exemplo é o de uma empresa iniciante, a Joe's Toothpaste, Inc., que comercializa o mais recente creme dental 100% orgânico, não-abrasivo, a base de tofu. Como há apenas um funcionário e esse funcionário fica muito ocupado procurando o melhor tofu que possa en-

contrar, o tubo não diz "Serviço de atendimento ao consumidor, ligue 1-800-TOFTOOT", mas sim "para reclamações ou comentários, visite nosso site na Web, www.toftoot.com". O site da Web tem todas as figuras chamativas de costume e uma área onde o consumidor pode digitar uma reclamação ou um comentário. A página é como a que aparece na Figura 28-1.

Figura 28-1 O que o consumidor encontra no endereço *http://www.toftoot.com/comment.html.*

Extrato do arquivo HTML

As partes principais do código HTML que gerou essa página são as seguintes:

```
<form method="post" action="http://toftoot.com/cgi-bin/feedback.py">
<ul><i>Please fill out the entire form:</i></ul>
<center><table width="100%" >
<tr>
    <td align="right" width="20%">Name:</td>
    <td>
        <input type="text" name="name" size="50" value="">
    </td>
</tr>
```

```
<tr>
    <td align="right">Email Address:</td>
    <td>
        <input type="text" name="email" size="50" value="">
    </td>
</tr>
<tr>
    <td align="right">Mailing Address:</td>
    <td>
        <input type="text" name="address" size="50" value="">
    </td>
</tr>
<tr>
    <td align="right">Type of Message:</td>
    <td>
        <input type="radio" name="type" checked
               value="comment">comment </input>
        <input type="radio" name="type"
               value="complaint">complaint</input>
    </td>
</tr>
<tr>
    <td align="right" valign="top">
        Enter the text in here:</td>
    <td><textarea name="text" rows="5" cols="50" value="">
        </textarea></td></tr>
<tr>
    <td></td>
    <td>
    <input type="submit" name="send" value="Send the feedback!">
    </td>
</tr>
</table></center>
</form>
```

Estamos supondo que você conheça o suficiente sobre CGI e HTML para acompanhar esta discussão. O código HTML gera a página da Web mostrada na Figura 28-1:

- A linha `form` especifica qual programa CGI deve ser ativado quando o formulário é enviado; especificamente, a URL aponta para um script chamado *feedback.py*.
- As tags `input` indicam os nomes dos campos no formulário (nome, endereço, email e texto, assim como o tipo). Os valores desses campos são o que o usuário digita, exceto quanto ao tipo, que recebe o valor `'comment'` ou o valor `'complaint'`, dependendo do botão de rádio selecionado pelo usuário.
- A tag `input type="submit"` é para o botão de envio, o qual realmente chama o script CGI.

Agora, vamos ver a parte interessante no que diz respeito ao Python: o processamento do pedido. Aqui está o programa *feedback.py* inteiro:

```
1    #!c:/python23/python.exe
2    import cgi, cgitb, os, sys, string, time
3    cgitb.enable()
```

```
4   def gush(data):
5       print """Content-type: text/html\n
6   <h3>Thanks, %(name)s!</h3>
7   Our customer's comments are always appreciated.
8   They drive our business directions, as well as
9   help us with our karma.
10  <p>Thanks again for the feedback!<p>
11  And feel free to enter more comments if you wish.""" % vars(data)
12      print "<p>"+10*" "+"--Joe."
13
14  def whimper(data):
15      print """Content-type: text/html\n
16  <h3>Sorry, %(name)s!</h3>
17  We're very sorry to read that you had a complaint"
18  regarding our product__We'll read your comments"
19  carefully and will be in touch with you."
20  <p>Nevertheless, thanks for the feedback.<p>""" % vars(data)
21      print "<p>"+10*" "+"--Joe."

22  def bail():
23      print """<h3>Error filling out form</h3>
24  Please fill in all the fields in the form.<p>
25  <a href="http://localhost/comment.html">
26  Go back to the form</a>"""
27      sys.exit()

28  class FormData:
29      """ A repository for information gleaned from a CGI form """
30      def __init__(self, form):
31          self.time = time.asctime()
32          for fieldname in self.fieldnames:
33              if fieldname not in form or form[fieldname].value == "":
34              bail()
35          else:
36              setattr(self, fieldname, form[fieldname].value)

37  class FeedbackData(FormData):
38      """ A FormData generated by the comment.html form. """
39      fieldnames = ('name', 'address', 'email', 'type', 'text')
40      def __repr__(self):
41          return "%(type)s from %(name)s on %(time)s" % vars(self)

42  DIRECTORY = r'C:\complaintdir'

43  if __name__ == '__main__':
44      sys.stderr = sys.stdout
45      form = cgi.FieldStorage()
46      data = FeedbackData(form)
47      if data.type == 'comment':
48          gush(data)
49      else:
50          whimper(data)

51      # Salva os dados no arquivo.
52      import tempfile, pickle
```

```
53      tempfile.tempdir = DIRECTORY
54      pickle.dump(data, open(tempfile.mktemp(), 'w'))
```

Claramente, a saída desse script depende da entrada, mas a saída com o formulário preenchido com os parâmetros mostrados na Figura 28-1 aparece na Figura 28-2.

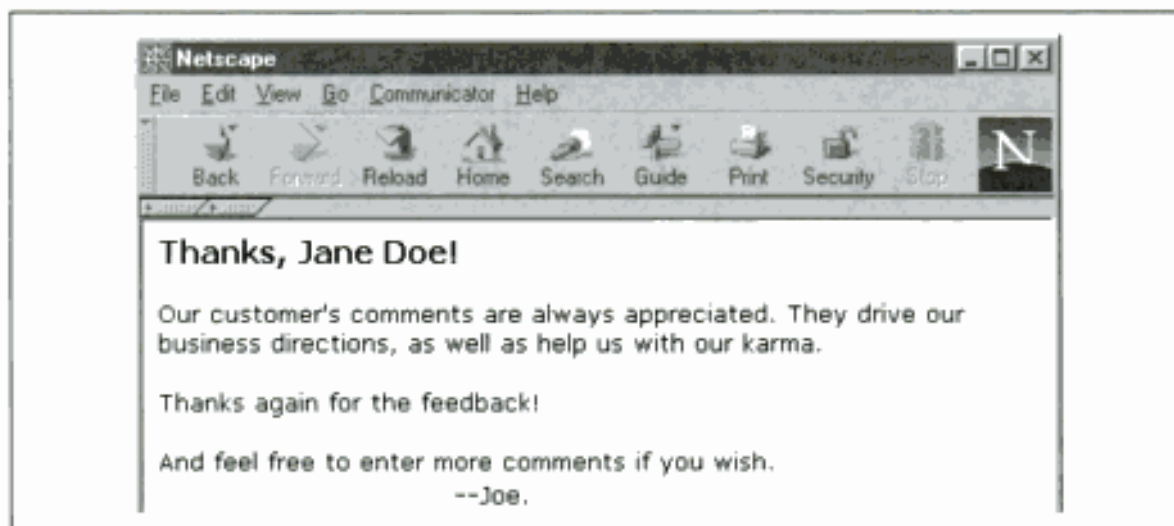



Figura 28-2 O que o usuário vê após pressionar o botão Send the feedback.

Como o script *feedback.py* funciona? Há alguns aspectos do script que são comuns a todos os programas CGI e eles estão destacados em negrito. Para começar, a primeira linha do programa precisa se referir ao executável em Python. Isso é um requisito do servidor da Web que estamos usando aqui e pode não se aplicar ao seu caso; mesmo que se aplique, a localização específica de seu programa em Python provavelmente será diferente desta. A segunda linha inclui importações de `cgi` e `cgitb`. O módulo `cgi` trata das partes difíceis do CGI, como analisar as variáveis de ambiente e manipular caracteres com escape. O módulo `cgitb`, que quer dizer "CGI Traceback", torna a depuração de aplicativos CGI muito mais fácil. Ele precisa ser habilitado (linha 3) para transformar as exceções do script em rastreamentos com uma impressão bonita. A documentação do módulo `cgi` descreve uma maneira muito simples e fácil de usá-lo. Para este exemplo, entretanto, principalmente porque vamos complementá-lo depois, o script é um pouco mais complicado do que o estritamente necessário.

Vamos apenas percorrer o código do bloco `if __name__ == '__main__'`, uma instrução por vez.* A primeira instrução (linha 44) redireciona o fluxo `sys.stderr` para qualquer que seja a saída padrão. Isso é feito para depuração, pois a saída do fluxo `stdout` em um programa CGI volta para o navegador da Web e o fluxo `stderr` vai para o log de erros do servidor, o qual pode ser mais difícil de ler simplesmente examinando a página da Web. Desse modo, se ocorrer uma exceção em tempo de execução, poderemos vê-la na página da Web e não adivinhar qual era ela.

A segunda linha (linha 45) é fundamental e faz todo o trabalho difícil da CGI: ela retorna um objeto tipo dicionário (chamado de objeto `FieldStorage`), cujas chaves são os nomes das variáveis preenchidas no formulário, enquanto o valor de cada campo no formulário pode ser obtido solicitando-se o atributo `value` das entradas no objeto `FieldStorage`. Parece complicado, mas tudo isso significa apenas que, para nosso formulário, o objeto formulário tem chaves

* Você se lembrará que essa instrução `if` é verdadeira somente quando o programa é executado como um script e não quando ele é importado. Os programas CGI se qualificam como scripts; portanto, o código no bloco `if` é executado quando esse programa é chamado pelo servidor da Web. O usaremos posteriormente como um script importado.

Agora temos uma infra-estrutura CGI básica implantada. Salvar os dados no arquivo é extraordinariamente fácil:

- Primeiro, definimos a variável DIRECTORY fora do teste `if`, pois a usaremos em outro script que importará este; portanto, queremos que ela seja definida mesmo que este script não seja executado como um programa.
- Na linha 52, importamos os módulos `tempfile` e `pickle`. O módulo `tempfile` sugere nomes de arquivo que não estão sendo usados; dessa forma, não precisamos nos preocupar com "conflitos" em algum esquema de geração de nome de arquivo. O módulo `pickle` nos permite dispor em série (isto é, salvar) qualquer objeto Python.
- Linha 53: a linha seguinte configura o atributo `tempdir` do módulo `tempfile` com o valor da variável DIRECTORY, que é onde queremos que nossos dados sejam salvos. Esse é um exemplo de personalização de um módulo existente por meio da modificação direta de seu espaço de nome, exatamente como modificamos o atributo `stderr` do módulo `sys`, anteriormente.
- Linha 54: a última linha realiza o salvamento real; ela abre o arquivo no modo de gravação, com um nome gerado pelo módulo `tempfile`, e despeja nele os dados da instância usando o módulo `pickle`. O módulo `pickle` é uma das preciosidades que torna o uso de uma linguagem de alto nível como o Python tão produtivo – *colocar em conserva* (*pickling*) significa pegar objetos Python arbitrários e convertê-los em fluxos de bytes em um formato que o Python sabe como "retirar da conserva". Como se trata de fluxos de bytes, eles podem ser gravados no disco, como estamos fazendo aqui, enviados por uma rede, armazenados ou transmitidos de alguma forma para posterior retirada da conserva. Não há necessidade de usar um formato de arquivo específico para esse aplicativo. Conforme você pode ver na linha 54, para salvar um objeto Python, basta passá-lo para `pickle.dump()`, pegar o resultado e colocá-lo em um arquivo. Agora o arquivo especificado contém uma instância colocada em conserva, que retiraremos da conserva na próxima seção.

INTERFACE UTILIZANDO COM: RELAÇÕES PÚBLICAS BARATAS

Neste ponto, temos um programa que é executado quando o usuário preenche o formulário de retorno e que grava nossas instâncias dos dados de retorno em arquivos no servidor. Usaremos esses dados para fazer duas coisas. Primeiro, um programa que é executado periodicamente (digamos, às 2:00 horas, toda noite)* examinará os dados salvos, descobrirá quais arquivos colocados em conserva correspondem a reclamações e imprimirá uma carta personalizada para o reclamante. O segundo uso que faremos desses dados é em um navegador de GUI para examinar as entradas de retorno armazenadas. Tudo isso parece complicado, mas você ficará surpreso de ver como isso é simples, usando as ferramentas corretas. O site da Web de Joe está em uma máquina Windows, mas outras plataformas funcionam de maneira semelhante.

Antes de falarmos sobre como escrever esse programa, uma palavra sobre a tecnologia que ele utiliza, a saber, o modelo COM (Component Object Model) da Microsoft. O COM é, dentre outras coisas, um padrão para interação entre programas que permite aos programas compatíveis com ele se comunicarem, acessar dados e executar comandos em outros programas também compatíveis. A grosso modo, o programa que faz a chamada é denominado cliente COM e o programa que executa é chamado servidor COM. O Microsoft Word é um

* A configuração desse tipo de programa, feito para executar regularmente de forma automática, é facilmente realizada na maioria das plataformas, usando, por exemplo, cron no Unix ou o Agendador de Tarefas no Windows NT/2000/XP.

deles e o utilizaremos aqui, pois é ótimo para escrever cartas, que é o que estamos fazendo. Felizmente, o Python também pode se tornar compatível com o modelo COM no Windows. Mark Hammond e Greg Stein tornaram disponível um conjunto de extensões do Python para Windows, chamado win32com, o qual permite que programas em Python façam quase tudo que você pode fazer com o modelo COM, a partir de qualquer outra linguagem. Você pode escrever clientes COM, servidores, hosts de script ActiveX, depuradores e muito mais, tudo em Python. Só precisamos fazer a primeira dessas tarefas, que também é a mais simples. As tarefas básicas que nosso programa gerador de cartas padronizadas precisa executar são:

1. Abrir todos os arquivos colocados em conserva no diretório apropriado e retirá-los da conserva para transformá-los novamente em objetos Python.
2. Para cada instância retirada da conserva, testar se o retorno é uma reclamação. Se for, descobrir o nome e o endereço da pessoa que preencheu o formulário e ir para a Etapa 3. Se não `for`, pula essa etapa.
3. Abrir um documento no Word contendo um modelo da carta que queremos enviar e preencher as partes apropriadas com informações personalizadas.
4. Imprimir o documento e fechá-lo.

Essa tarefa é quase tão simples quanto expressar em Python com o módulo win32com. Aqui está um programa chamado *formletter.py*:

```
from win32com.client import gencache, constants
WORD = 'Word.Application'
False, True = 0, -1

class Word:
    def __init__(self):
        self.app = gencache.EnsureDispatch(WORD)
    def open(self, doc):
        self.app.Documents.Open(FileName=doc)
    def replace(self, source, target):
        self.app.Selection.HomeKey(Unit=constants.wdLine)
        find = self.app.Selection.Find
        find.Text = "%"+source+"%"
        find.Execute()
        self.app.Selection.TypeText(Text=target)
    def printdoc(self):
        self.app.Application.PrintOut()
    def close(self):
        self.app.ActiveDocument.Close(SaveChanges=False)

def print_formletter(data):
    word.open(r"h:\David\Book\tofutemplate.doc")
    word.replace("name", data.name)
    word.replace("address", data.address)
    word.replace("firstname", data.name.split()[0])
    word.printdoc()
    word.close()
if __name__ = '__main__':
    import os, pickle
    from feedback import DIRECTORY, FeedbackData
    word = Word()
    for filename in os.listdir(DIRECTORY):
```

variável de instância `app`. Agora, existem duas maneiras pelas quais o código subseqüente pode usar esse servidor: envio dinâmico e envio não-dinâmico. No envio dinâmico, quando o programa está em execução, o Python não sabe qual é a interface com o servidor COM (neste caso, o Microsoft Word). Isso não é problema, pois o modelo COM permite que o Python consulte o servidor e determine o número e os tipos de argumentos esperados por cada função.

Para explicarmos os métodos da classe `Word`, vamos começar com um possível documento modelo, mostrado na Figura 28-3, para que possamos ver o que precisa ser feito nele para personalizá-lo.

JOE'S TOOTHPASTE, INC

December 1, 2003

%name%
%address%

Dear %firstname%:

Thank you for filling out the feedback form on our website (www.toftoot.com). We're sorry to hear of your complaint, and we'll do our best to remedy it in the future. As a token of our deepest apologies, please accept a coupon for a free tube of our newest toothpaste, the "Firm Tofu Plus Toothpaste", enriched with vitamins A, B and Z12.

Sincerely,

Joe Smith
CEO, CFO and COO, Joe's Toothpaste

FREE COUPON

Coupon for one (1) free tube of Firm Tofu Plus Toothpaste (5 oz)
Available at all WackyStores ™

SUITE 101, 22 MAIN ST • TOFUTOWN, CA 04122
PHONE: 415-555-1212 • FAX: 415-555-1211

Figura 28-3 Carta-modelo de Joe para os reclamantes.

Conforme você pode ver, trata-se de um documento bem normal, com exceção de algum texto entre sinais %. Usamos essa notação apenas para tornar fácil para um programa encontrar as partes que precisam de personalização, mas qualquer outra técnica também poderia funcionar. Para usar esse modelo, precisamos abrir o documento, personalizá-lo, imprimi-lo e fechá-lo. A abertura é feita pelo método `open` da classe `Word`. A impressão e o fechamento são feitos de maneira análoga. Para personalizar, substituímos o texto `%name%`, `%firstname%` e `%address%` pelas strings apropriadas. É isso que o método replace da classe `Word` faz (não abordaremos como descobrimos qual deve ser a seqüência de chamadas exata; consulte o quadro "Descobrindo sobre as interfaces COM" para ver os detalhes).

Colocando tudo isso em funcionamento, o programa, quando executado, gera na saída um texto como o seguinte:

```
C:\Programs> python formletter.py
Printing letter for John Doe.
```

```
Got comment from Your Mom, skipping printing.
Printing letter for Susan B. Anthony.
```

e imprime duas cartas personalizadas, prontas para serem enviadas pelo correio. Note que o programa Word não aparece na área de trabalho; por padrão, os servidores COM são invisíveis, de modo que o Word atua apenas nos bastidores. Se o Word estiver correntemente ativo na área de trabalho, cada etapa será visível para o usuário.

UM EDITOR DE GUI BASEADO EM TKINTER PARA GERENCIAR DADOS DE FORMULÁRIO

Vamos recapitular: escrevemos um programa CGI (*feedback.py*) que pega a entrada de um formulário da Web e armazena as informações no disco em nosso servidor. Então, escrevemos um programa (*formletter.py*) que pega alguns desses arquivos e gera pedidos de desculpa para quem os merece. A próxima tarefa é construir um programa para permitir que um ser humano veja os comentários e as reclamações, usando o kit de ferramentas Tkinter para construir um navegador de GUI para esses arquivos.

O kit de ferramentas Tkinter é uma interface específica do Python para uma biblioteca de GUI não pertencente à linguagem, chamada Tk. Tk é o kit de ferramentas de GUI mais comumente escolhido pelos programadores de Python, porque fornece GUIs de aparência profissional dentro de um sistema muito fácil de usar e porque a interface Python/Tk acompanha as distribuições de Python. As interfaces que ele gera não são exatamente como as do Windows, do Mac ou de qualquer kit de ferramentas Unix, mas são muito parecidas com cada uma delas e o mesmo programa em Python funciona em todas essas plataformas, algo impossível com

Descobrindo sobre as interfaces COM

Como você descobre quais são os vários métodos e atributos dos objetos COM? Em geral, os objetos COM são exatamente como qualquer outro programa; eles devem vir com documentação. Entretanto, no caso dos objetos COM, é bastante possível ter o software sem a documentação, simplesmente porque, como no caso do Word, é possível usar o Word sem precisar programá-lo. Há três estratégias disponíveis para você, se quiser explorar uma interface COM:

- Encontre ou adquira a documentação; alguns programas COM têm sua documentação disponível na Web ou em forma impressa.
- Use um navegador COM para explorar os objetos. O Pythonwin (parte das extensões win32all para Python no Windows), por exemplo, vem com uma ferramenta de navegador COM que permite explorar a complexa hierarquia de objetos COM. Ela não é muito mais do que uma listagem dos objetos e funções disponíveis, mas às vezes isso é tudo que você precisa. Ferramentas de desenvolvimento, como o Visual Studio da Microsoft, também acompanham os navegadores COM.
- Use outra ferramenta para descobrir o que está disponível. Para o exemplo, usamos simplesmente o recurso gravador de macros do Microsoft Word para produzir um script VBA (Visual Basic for Applications), que é muito simples de transformar em Python. As macros tendem a ser programas de muito pouca inteligência, significando que o recurso de gravação de macro não consegue captar o fato de que talvez você quisesse fazer algo dez vezes e, assim, apenas grava a mesma ação várias vezes. Mas elas funcionam bem para descobrir que o equivalente de selecionar o item Imprimir do menu Arquivo é "dizer" `ActiveDocument.PrintOut()`.

O bloco do loop `for`, iterando pelo atributo `fieldnames` da variável `dataclass` (a classe `fieldnames` da classe `FeedbackData`), descobre quais variáveis estão nos dados da instância e, para cada uma, chama o método `add_variable` da classe `FormEditor`, pega o valor retornado e o coloca em uma variável de instância. Isso é equivalente, em nosso caso, a:

```
    ...
self.name = self.add_variable(root, 'name')
self.email = self.add_variable(root, 'email')
self.address = self.add_variable(root, 'address')
self.type = self.add_variable(root, 'type')
self.text = self.add_variable(root, 'text')
```

Entretanto, a versão no exemplo de código é melhor, pois a lista de nomes de campo já está disponível para o programa e redigitar tudo normalmente é um indicativo de um projeto malfeito. Além disso, não há nada a respeito de `FormData` que seja específico para nossos formulários. Ela pode ser usada para navegar em qualquer instância de uma classe que defina uma variável `fieldnames`. Tornar o programa genérico, dessa forma, torna-o mais provável de ser reutilizado em outros contextos, para outras tarefas.

Terminando com o método `__init__`, vemos que dois botões finalizam o layout gráfico da janela, cada um associado a um comando executado quando recebe um clique de mouse. Um é o método `delentry`, que exclui a entrada corrente, e o outro é uma função de recarregamento que relê os dados no diretório de armazenamento.

Finalmente, os dados são carregados por uma chamada para o método `load_data`. Vamos descrevê-lo assim que tivermos terminado com as chamadas que configuram elementos de janela; a saber, `add_variable` e `add_button`.

`add_variable` cria dois elementos de janela `Label` na mesma linha. O primeiro exibe o nome do campo e o segundo conterá o valor do campo correspondente na entrada selecionada na caixa de listagem:

```
def add_variable(self, root, varname):
    Label(root, text=varname).grid(row=self.row, column=0, sticky=E)
    value = Label(root, text=' ', background='gray90',
                  relief=SUNKEN, anchor=W, justify=LEFT)
    value.grid(row=self.row, column=1, sticky=E+W)
    self.row = self.row + 1
    return value
```

`add_button` é mais simples, pois precisa criar apenas um elemento de janela:

```
def add_button(self, root, row, column, text, command):
    button = Button(root, text=text, command=command)
    button.grid(row=row, column=column, sticky=E+W, padx=5, pady=5)
```

A função `load_data` é chamada quando o botão `Refresh` é selecionado. Antes de carregar os dados do arquivo colocado em conserva, ela primeiro limpa todo o conteúdo da caixa de listagem (a lista de itens gráfica), que possivelmente correspondem a dados desatualizados, e reconfigura o atributo `items` (que é uma lista do Python que conterá referências para as instâncias de dados reais). O loop que preenche os atributos `listbox` e `items` é muito parecido com aquele usado por *formletter.py*:

```
def load_data(self):
    self.listbox.delete(0,END)
    self.items = []
```

ela veio. Usamos essa informação para excluir o arquivo, antes de solicitar um recarregamento; a caixa de listagem é atualizada automaticamente:

```
def delentry(self):
    os.remove(os.path.join(self.storagedir, self.selection._filename))
    self.load_data()
```

Esse programa provavelmente é o mais difícil de entender de qualquer outro deste livro, simplesmente porque ele usa extensivamente a complexa e poderosa biblioteca Tkinter. Existe documentação para Tkinter, assim como para a própria Tk.

- A documentação mais completa é a de Frederik Lundh, disponível na Web no endereço *http://www.pythonware.com/library/tkinter/introduction/index.htm.*
- Vários livros abordam o Tkinter. John Grayson escreveu um livro dedicado à programação de Tkinter, *Python and Tkinter Programming* (Manning Publications). O livro *Programming Python* da O'Reilly também tem uma ampla cobertura do Tkinter, incluindo uma seção de 260 páginas dedicada aos elementos de janela básicos e exemplos de programa completos. Finalmente, o livro *Python in a Nutshell* (O'Reilly) tem uma documentação concisa, abordando a maior parte do Tkinter.
- O New Mexico Institute of Mining and Technology criou seu próprio manual de Tkinter, com 84 páginas. Ele está disponível nos formatos PDF e PostScript, no endereço *http://www.nmt.edu/tcc/help/lang/python/docs.html*. Procure sob "Locally written documentation".
- Além disso, veja a seção sobre Tkinter no site na Web do Python, no endereço *http://www.python.org/topics/tkinter/.*

Considerações de projeto

Considere o script CGI *feedback.py* e o programa de GUI *FormEditor.py* como duas maneiras diferentes de manipular um conjunto de dados comum (as instâncias colocadas em conserva no disco). Quando você deve usar uma interface baseada na Web e quando deve usar uma GUI? A escolha deve ser baseada em dois fatores:

- É fácil implementar a funcionalidade necessária em determinado modelo?
- Que software você pode exigir que o usuário instale para acessar ou modificar os dados?

O front-end da Web é muito conveniente para os casos onde a complexidade dos requisitos de manipulação de dados é pequena e onde é mais importante que os usuários possam trabalhar no programa do que o programa seja completo. Por outro lado, construir um programa real com um kit de ferramentas de GUI possibilita o máximo de flexibilidade, ao custo de ter de ensinar ao usuário sobre como usá-lo e/ou instalar programas específicos. Uma razão do sucesso do Python entre os programadores experientes é que ela permite projetar programas sobre bases fundamentadas, em oposição a obrigá-los a usar um único tipo de modelo de programação, apenas porque era isso que o projetista da linguagem tinha em mente.

JYTHON: A FELIZ UNIÃO DE PYTHON E JAVA

Jython é uma versão do Python escrita inteiramente em Java. O Jython é muito estimulante tanto para a comunidade do Python como para a comunidade da linguagem Java. Os usuários de Python ficam felizes porque seu conhecimento atual pode ser aplicado em projetos baseados na linguagem Java; os programadores de Java ficam felizes porque podem usar a lingua-

gem de script Python como uma maneira de controlar seus sistemas Java, testar bibliotecas e aprender sobre as bibliotecas Java, trabalhando em um ambiente interativo poderoso.

O Jython está disponível no endereço *http://www.jython.org*, com termos de licença e distribuição semelhantes aos do CPython (que é como a implementação de referência do Python é chamada, quando contrastada com o Jython).

A instalação de Jython inclui várias partes:

- *jython*, que é o equivalente do programa python usado neste livro.
- *jythonc*, que pega um programa em Jython e o compila em arquivos de classe Java. Os arquivos de classe Java resultantes podem ser usados como qualquer arquivo de classe normal; por exemplo, como applets, servlets ou beans.
- Um conjunto de módulos que fornece ao usuário de Jython a ampla maioria dos módulos presentes na biblioteca padrão do Python.
- Alguns programas demonstrando vários aspectos da programação em Jython.

Usar Jython é muito parecido com usar Python:

```
~/book> jython
Jython 2.1 on java1.3.1_03 (JIT: null)
Type "copyright", "credits" or "license" for more information.
>>> 2 + 3
5
```

Na verdade, o Jython funciona de forma quase idêntica ao CPython. Para ver uma listagem atualizada das diferenças entre os dois, consulte o endereço *http://www.jython.org/doc/differences.html*. As diferenças mais importantes são:

- Atualmente, o Jython é mais lento do que o CPython. O quanto ele é mais lento depende do código de teste usado e da Máquina Virtual Java que o Jython esteja usando.
- Alguns módulos internos ou de biblioteca não estão disponíveis para o Jython. Por exemplo, a chamada de `os.system()` ainda não está implementada, pois isso é difícil, dada a interação da linguagem Java com o sistema operacional subjacente. Além disso, alguns dos maiores módulos de extensão, como o modelo de GUI Tkinter, não estão disponíveis, pois as ferramentas subjacentes (o kit de ferramentas Tk/Tcl, no caso do Tkinter) não estão disponíveis em Java.

O Jython fornece aos programadores de Python acesso às bibliotecas Java

Contudo, a diferença mais importante entre o Jython e o CPython é que o primeiro oferece ao programador de Pyhon acesso transparente às bibliotecas Java. Considere o programa a seguir, *jythondemo.py*, cuja saída aparece na Figura 28-5.

```
from pawt import swing
import java

def exit(e): java.lang.System.exit(0)

frame = swing.JFrame('Swing Example', visible=1)
button = swing.JButton('This is a Swinging example!', actionPerformed=exit)
frame.contentPane.add(button)
frame.pack()
```

Um aplicativo Jython/Swing real: grapher.py

O programa *grapher.py* (cuja saída aparece na Figura 28-6) permite que os usuários explorem graficamente o comportamento de funções matemáticas. Ele também é baseado no kit de ferramentas de GUI Swing. Existem dois elementos de tela para entrada de texto, nos quais o código Python deve ser inserido. O primeiro é um programa em Python arbitrário, chamado antes que a função seja desenhada; ele importa os módulos necessários e define todas as funções que possam ser necessárias no cálculo do valor da função. A segunda área de texto (rotulada como `Expression:`) deve ser uma expressão em Python (como em `sin(x)`) e não uma instrução. Ela é chamada para cada ponto de entrada, com o valor da variável `x` configurado com a coordenada horizontal.

O usuário pode controlar se vai desenhar um gráfico de linhas ou um gráfico de preenchimento, o número de pontos a representar no gráfico e a cor do gráfico. Finalmente, o usuário pode salvar configurações no disco e recarregá-las posteriormente (usando o módulo `pickle`). Aqui está o programa *grapher.py*:

```
from pawt import swing, awt, colors, GridBag
RIGHT = swing.JLabel.RIGHT
APROVE_OPTION = swing.JFileChooser.APPROVE_OPTION
import java.io
import pickle, os

default_setup = """from math import *
def squarewave(x,order):
```

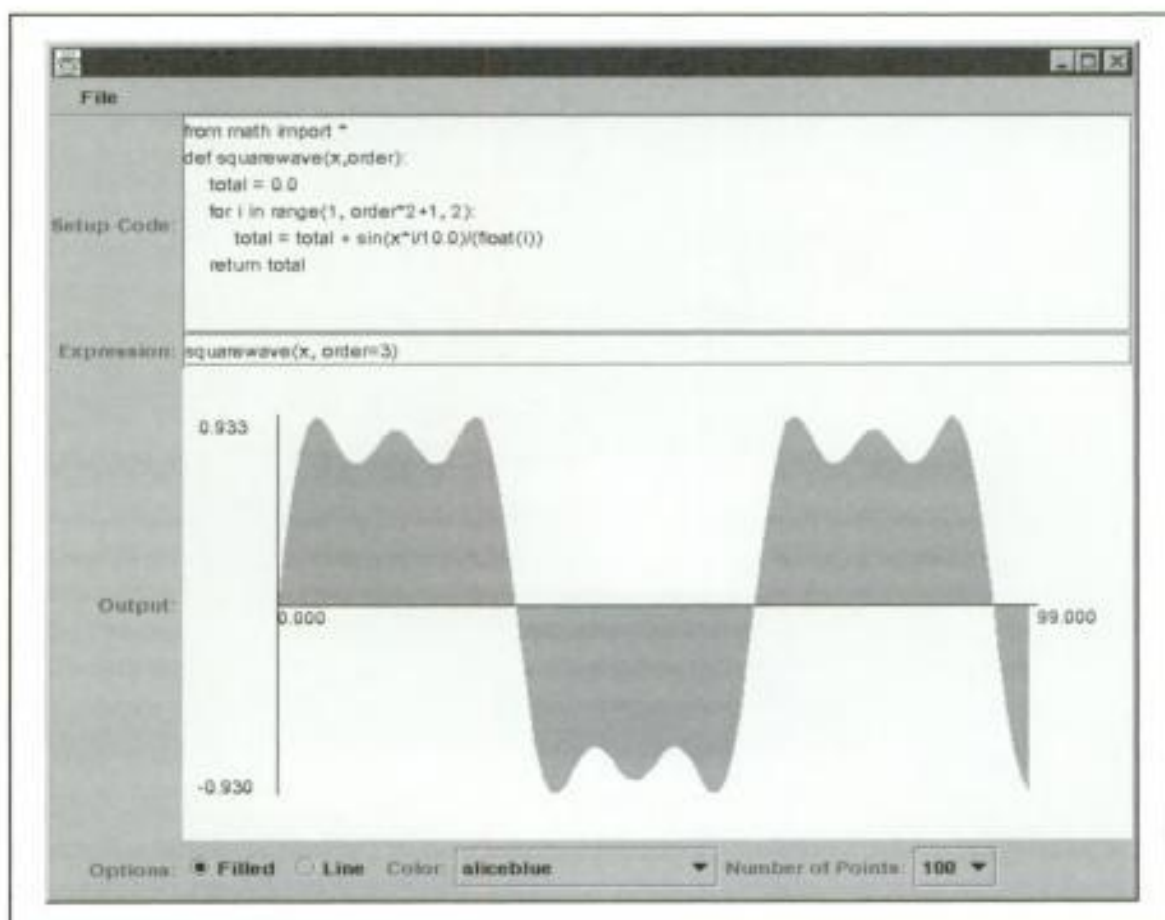



Figura 28-6 Saída do programa *grapher.py*.

```
        bag = GridBag(self.frame.contentPane, fill='BOTH',
                      weightx=1.0, weighty=1.0)
        bag.add(swing.JLabel("Setup Code: ", RIGHT))
        bag.addRow(swing.JScrollPane(self.execentry), weighty=10.0)
        bag.add(swing.JLabel("Expression: ", RIGHT))
        bag.addRow(self.evalentry, weighty=2.0)
        bag.add(swing.JLabel("Output: ", RIGHT))
        bag.addRow(self.chart, weighty=20.0)
        bag.add(swing.JLabel("Options: ", RIGHT))
        bag.addRow(optionsPanel, weighty=2.0)
        self.update(None)
        self.frame.visible = 1
        self.frame.size = self.frame.getPreferredSize()

        self.chooser = swing.JFileChooser()
        self.chooser.currentDirectory = java.io.File(os.getcwd())

    def do_save(self, event=None):
        self.chooser.rescanCurrentDirectory()
        returnVal = self.chooser.showSaveDialog(self.frame)
        if returnVal == APROVE_OPTION:
            object = (self.execentry.text, self.evalentry.text,
                      self.chart.style,
                      self.chart.color.RGB,
                      self.colorname.selectedIndex,
                      self.numelements)
        file = open(os.path.join(self.chooser.currentDirectory.path,
                    self.chooser.selectedFile.name), 'w')
        pickle.dump(object, file)
        file.close()

    def do_load(self, event=None):
        self.chooser.rescanCurrentDirectory()
        returnVal = self.chooser.showOpenDialog(self.frame)
        if returnVal == APROVE_OPTION:
            file = open(os.path.join(self.chooser.currentDirectory.path,
                        self.chooser.selectedFile.name))
            (setup, each, style, color,
             colorname, self.numelements) = pickle.load(file)
            file.close()
            self.chart.color = java.awt.Color(color)
            self.colorname.selectedIndex = colorname
            self.chart.style = style
            self.execentry.text = setup
            self.numpoints.selectedIndex = self.sizes.index(self.numelements)
            self.evalentry.text = each
            self.update(None)

    def do_quit(self, event=None):
        import sys
        sys.exit(0)

    def set_color(self, event):
        self.chart.color = getattr(colors, event.item)
        self.chart.repaint()
```

```
        def set_numpoints(self, event):
            self.numelements = event.item
            self.update(None)

        def set_filled(self, event):
            self.chart.style = 'Filled'
            self.chart.repaint()

        def set_line(self, event):
            self.chart.style = 'Line'
            self.chart.repaint()

        def update(self, event):
            context = {}
            exec self.execentry.text in context
            each = compile(self.evalentry.text, '<input>', 'eval')
            numbers = [0]*self.numelements
            for x in xrange(self.numelements):
                context['x'] = float(x)
                numbers[x] = eval(each, context)
            self.chart.data = numbers
            if self.chart.style == 'Line':
                self.line.setSelected(1)
            else:
                self.filled.setSelected(1)
                self.chart.repaint()

GUI()
```

A lógica desse programa é muito simples e os nomes de classe e de método tornam fácil acompanhar o fluxo de controle. A maior parte desse programa poderia ter sido escrito em código Java bastante parecido (mas muito mais longo). Entretanto, as partes em negrito mostram o poder de se ter o Python disponível: no início do módulo, são definidos valores padrão para os elementos de janela textuais `Setup` e `Expression`. O primeiro importa as funções do módulo `math` e define uma função chamada `squarewave`. O último especifica uma chamada para essa função, com um parâmetro de ordem específico (à medida que esse parâmetro cresce, o gráfico resultante fica cada vez mais parecido com uma onda quadrada; daí, o nome da função – `squarewave` significa onda quadrada, em inglês) Se você tiver Java Swing e Jython instalados, veja outras possibilidades para os dois elementos de janela textuais `Setup` e `Expression`.

A principal vantagem de usar Jython, em vez de Java, nesse exemplo, está no método `update`: ele simplesmente chama a instrução `exec` padrão do Python, com o código de `Setup` como argumento, e então chama `eval` com a versão compilada do código de `Expression` para cada coordenada. O usuário fica livre para usar qualquer código Python nesses elementos de janela textuais!

O Jython ainda é um trabalho em andamento; seus desenvolvedores estão constantemente refinando a interface entre Python e Java, otimizando-a e acompanhando as atualizações do Python. O Jython, sendo a segunda implementação do Python, também está obrigando Guido van Rossum a decidir quais aspectos do Python são a base da linguagem e quais são recursos de sua implementação.

EXERCÍCIOS

A maioria dos tópicos deste capítulo não são realmente bons assuntos para exercícios, sem primeiro abordar os modelos que eles cobrem. Contudo, duas coisas podem ser feitas com o conhecimento que você já tem:

1. *Imitando a Web*. Talvez você não tenha um servidor da Web executando, o que torna o uso dos programas *formletter.py* e *FormEditor* difícil, pois eles usam dados gerados pelo script CGI. Como exercício, escreva um programa que crie arquivos com as mesmas propriedades daqueles criados pelo script CGI.
2. *Limpeza*. Há um problema sério no programa *formletter.py*: a saber, se ele for executado à noite, toda reclamação irá fazer com que uma carta seja impressa. Isso acontecerá todas as noites, pois não há nenhum mecanismo para indicar que uma carta foi gerada e que mais nenhuma precisa ser, a respeito dessa reclamação específica. Corrija esse problema.
3. *Adicionando traçado de gráfico paramétrico no programa grapher.py*. Modifique o programa *grapher.py* para permitir que o usuário especifique expressões que retornam valores de *x* e *y*, em vez da solução corrente que imprime apenas *y*. Por exemplo, o usuário poderá escrever a expressão `widget: sin(x/3. 1),cos(x/6.15)` (observe a vírgula: isso é uma tupla!) e obter uma figura como a que aparece na Figura 28-7.

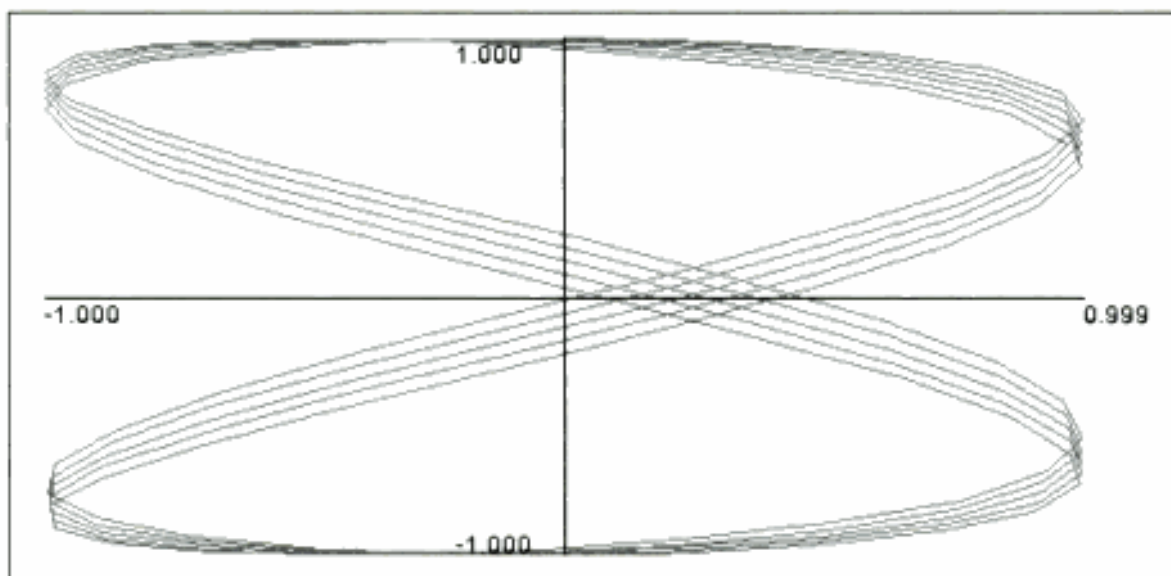
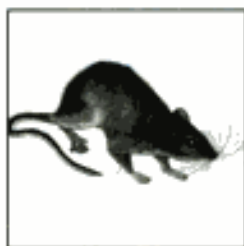



Figura 28-7 Saída do Exercício 3.

29

Recursos do Python

Assim como as linguagens naturais, as linguagens de programação desenvolvem comunidades fortemente integradas – os falantes do mesmo idioma têm uma afinidade natural entre si. As linguagens de programação, sendo o resultado de escolha individual, em vez de um acaso do nascimento, levam a sentimentos mais fortes de afinidade do que poderia parecer razoável para o que alguns vêem como puro assunto tecnológico. As linguagens de código-fonte aberto, que nunca são escolhidas por causa de uma campanha de marketing, mas sim após um processo de deliberação e comparação, parecem provocar ainda mais entusiasmo (alguns poderiam até dizer fanatismo) em seus usuários. Este capítulo fala sobre a comunidade que se define como "A Comunidade Python", desde o santuário secreto de pessoas que sonham com o Python diariamente até o usuário ocasional.

Como é muito fácil escrever código Python que pode ser compartilhado, grande parte dessa comunidade partilha seu entusiasmo e seu trabalho, muito freqüentemente na forma de ainda mais software livre. O efeito bola-de-neve resultante (ou, para usar um termo mais em voga, o "efeito rede") torna facílimo escrever programas grandes, em comparação a muitas outras opções de linguagem. Este capítulo mostrará algumas das ofertas de terceiros mais valiosas, desde pequenos módulos, interfaces, bibliotecas de sistema operacional, até repositórios de módulo.

CAMADAS DA COMUNIDADE

Nesta seção, discutiremos as várias camadas da comunidade, desde o núcleo de programadores muito sérios que implementam o interpretador oficial do Python, passando pela Python Software Foundation, grupos de interesse especial e grupos de usuários, até o amplo espectro de participantes, conhecidos como "python-list". Você provavelmente verá que pertence a uma ou mais dessas comunidades e talvez queira visitar algumas que ainda não conhece. Independente de onde você opte por se instalar, bem-vindo à nossa comunidade!

O núcleo

Ao contrário de muitas comunidades de linguagem, o mundo Python tem um centro muito claro. Esse centro tem se desenvolvido com o passar dos anos, com Guido van Rossum como núcleo permanente, circundado fisicamente por um grupo leal chamado "Pythonlabs" e cir-

cundado virtualmente pelo "grupo python-dev". O Pythonlabs é composto de alguns desenvolvedores de Python principais (Tim Peters, Barry Warsaw, Jeremy Hylton e Fred Drake), que foram recrutados por Guido para trabalharem com ele no Python e em projetos relacionados, primeiramente no BeOpen.com e depois na Zope Corporation. Junto com Guido, geralmente têm sido eles que fazem as alterações mais radicais nos mecanismos internos do Python, embora tenha havido algumas exceções notáveis por parte de outros colaboradores.

O Python é grande demais para que poucas pessoas gerenciem e desenvolvam com a rapidez que seus usuários gostariam (especialmente porque o Python é apenas um trabalho de meio-expediente, mesmo para Guido). Um elenco de apoio, geralmente referido como povo do python-dev, de acordo com a lista de distribuição que ancora as discussões, está disponível para ajudar em discussões sobre projeto, implementação, teste e, principalmente, debater (entretanto, tudo de boa fé).

Contudo, para a maioria dos usuários de Python, o trabalho do Pythonlabs e do Python-dev é gratamente reconhecido, mas um tanto misterioso. Muito mais pessoas vivem em uma das camadas externas do domínio Python, virtuais ou físicas (ou, espera-se, em ambas). Voltaremos às camadas tecnicamente "profundas" posteriormente – entretanto, é bem mais razoável aprender sobre uma comunidade a partir de uma agência de turismo do que a partir do comitê de planejamento urbano.

Grupos de usuários locais

Embora a maioria das comunicações relacionadas ao Python ocorra na Internet, é interessante fundamentar os nomes com faces e acentos, e ter uma idéia das personalidades do mundo real por trás das pessoas on-line. Há duas maneiras excelentes de fazer isso, cada uma com suas próprias vantagens: grupos de usuários e conferências.

Talvez você tenha um grupo de usuários do Python ou da Zope local em sua vizinhança. Existe uma lista no endereço *http://www.python.org/UserGroups.html*, mas verifique em sua sociedade de computação local, grupo de usuários de Linux ou outra organização assim, para saber se existem informações mais atualizadas. Se ainda não houver uma, são boas as chances de que exista demanda reprimida para isso, e elas são fáceis de organizar. Essa é uma excelente ocasião para aprender mais sobre o Python, descobrir quem mais está usando Python nas imediações (o que pode ser útil no momento de enviar um currículo) e conhecer pessoas com quem você compartilha pelo menos um interesse. Os dois autores deste livro visitam grupos de usuários locais regularmente.

Conferências e workshops

Se você toma ou não uma cerveja de vez em quando com usuários de Python locais, o encorajamos a participar de conferências sobre Python. Existem várias conferências regulares, todas interessantes.

A conferência mais antiga, a International Python Conference (IPC), agora faz parte da O'Reilly Open Source Convention (OSCON) (consulte o endereço *http://conferences.oreillynet.com* para obter informações). Com o passar dos anos, a IPC passou de uma pequena reunião informal para uma grande assembléia, com muitos participantes, onde se misturam especialistas em Python, entusiastas e iniciantes de todo o mundo. Para muitos, ela é a única ocasião para falar com seus colegas de Python na vida real e vale a pena participar.

Uma nova conferência, a PyCon, começou recentemente. Destinada a complementar o caminho da OSCON, ela é uma conferência de baixo custo para aficionados, enfatizando discus-

sões técnicas e precedendo os aspectos mais comerciais de feiras, como saguão de exposições e almoços elegantes. Quando não estou escrevendo este capítulo, eu (David) estou ajudando a planejar o segundo evento da PyCon, que promete ter apresentações de alta qualidade.

As duas conferências mencionadas acontecem nos EUA. Uma alternativa para alguns é ir para a Europa e participar das conferências sobre Python que acontecem por lá: EuroPython (*http://www.europython.org*) e a Python UK Conference (*http://www.pythonuk.org*). Mais conferências podem ter surgido para suprir o pessoal. Consulte o endereço *http://www.python.org/workshops/* para ver atualizações.

Onde obter ajuda

Sendo tão fácil de aprender como é o Python, nenhum livro ou site na Web pode dar todas as respostas. Se você tiver uma pergunta relacionada ao Python, existem algumas possibilidades de suporte gratuito e pago. Informações sobre todas as listas de distribuição mencionadas a seguir, incluindo como se inscrever, estão disponíveis no endereço *http://www.python.org/psa/MailingLists.html*.

Python-help

Se você tem uma pergunta sobre o Python, para a qual não consegue obter resposta através dos meios usuais, pode enviar um email para *Python-help@python.org*. Seu email será enviado para um grupo de voluntários dispersos pelo mundo, que farão o melhor possível para responder. Seja o mais detalhado possível em suas perguntas, copie e cole no email o seu código e os erros que recebe (em oposição às paráfrases freqüentemente confusas) e seja paciente enquanto espera por uma resposta. Você verá que a python-help pode ser um recurso muito útil.

Python-tutor

Embora a python-help forneça ajuda particularizada, a python-tutor é uma lista de distribuição para as pessoas que estão procurando ajuda nos primeiros estágios do aprendizado sobre Python ou mesmo sobre programação usando Python. Trata-se de uma lista amistosa, onde nenhuma pergunta é considerada básica demais e ainda é de volume muito baixo. Consulte o endereço *http://www.python.org/psa/MailingLists.html#tutor* para ver os detalhes.

Python-list

Python-list é o pessoal do desenvolvimento do Python. Ela está disponível como lista de distribuição ou como o newsgroup da Usenet *comp.lang.python*. Com o passar dos anos, ela se tornou um destino *muito* popular, com milhares de mensagens por mês. As conversas tendem a ser sobre Python, embora ocorram algumas excursões interessantes sobre os assuntos mais estranhos. É um bom lugar para aparecer na rede, com pessoas que pensam da mesma maneira. A Python-list define a face pública da comunidade Python de modo geral.

Grupos de interesse especial

Todos os recursos mencionados até agora eram gerais. Dois tipos de grupos de discussão mais focalizados também precisam ser mencionados. O primeiro, e mais essencial, é a discussão em torno da implementação da linguagem em si, em uma lista chamada python-dev. Além disso, alguns tópicos especializados naturalmente levam a discussões longas, as quais podem ficar muito técnicas, mas ainda permanecem dentro de um domínio bem definido. Exemplos

típicos desse tipo de grupo de interesse especial são as discussões destinadas a definir bibliotecas especializadas que devem ser adicionadas no Python, como aquelas para processamento científico, interfaces de banco de dados, suporte para XML ou Unicode. As pessoas interessadas em trabalhar nessas áreas, normalmente formam uma lista de distribuição de grupo(s) de interesse especial para coordenar suas atividades, obter acesso a uma área do endereço *http://www.python.org* para publicar seus resultados e, quando tiverem chegado a um acordo interno, influenciar a grande comunidade Python (e Guido, em particular) para incluir uma funcionalidade específica (como fez a GID Numeric para algumas alterações sintáticas) ou apenas relatar um acordo (como fez a GID Database para anunciar sua definição do padrão DB-API). Atualmente, muitas discussões satélites ocorrem em outras listas de distribuição que não são SIGs "oficiais", embora elas tendam a não dar muito retorno para o núcleo da linguagem. Informações sobre SIGs estão sempre disponíveis no endereço *http://www.python.org/sigs*.

python-dev

Além daquelas poucas conversas "físicas" que acontecem na região de Reston, VA, EUA (onde os membros da equipe do Pythonlabs estão localizados), diariamente, discussões sobre o desenvolvimento do Python em si ocorrem em uma lista de distribuição chamada python-dev. O número de pessoas que participam dessa lista é consideravelmente maior (dezenas de colaboradores ativos, provavelmente milhares de leitores regulares) e fornece um fórum muito bom para as pessoas que já sabem a maior parte do que há para saber sobre o Python de hoje e querem participar do desenvolvimento do Python de amanhã. Se você estiver curioso, pode se inscrever na lista de distribuição (*http://mail.python.org/mailman/listinfo/python-dev*), lê-la por meio de uma interface de correio/Web (por exemplo, *http://aspn.activestate.com/ASPN/Mail/Browse/Threaded/python-dev/*) ou por meio de interfaces de correio/notícias (veja, por exemplo, *http://www.gmane.org*). Entretanto, a maioria dos leitores deste livro achará as discussões na python-dev detalhadas e técnicas demais. Para ter uma idéia do que está mantendo os desenvolvedores do Python básico ocupados, enquanto estão acordados, é melhor você ler os resumos editados que são postados periodicamente na python-dev e na python-list e que são arquivados no endereço *http://www.python.org/dev/summary/* (você também pode pedir para recebê-los por email). É importante lembrar que a Python-dev é um *grupo de trabalho*. Se você é iniciante no Python, é bem-vindo para ler, mas provavelmente não deve postar, exceto em circunstâncias muito raras. Definitivamente, você *não* deve pedir ajuda na python-dev – conforme vimos, existem muitos outros lugares mais apropriados para pedir ajuda. Analogamente, pedidos de nova sintaxe ou novos recursos são, de modo geral, inadequados na python-dev.

Novas fontes

Se você é ocupado demais para acompanhar a python-list regularmente, existem várias opções para certificar-se de saber de todos os novos desenvolvimentos no mundo Python.

Normalmente, os anúncios importantes são enviados para a python-list e para a python-announce, que é uma lista moderada. Esse é o principal fórum para divulgar novos pacotes ou módulos do Python, novas conferências e outros anúncios importantes de interesse geral para a comunidade.

Várias pessoas relatam desenvolvimentos no Python, como Frederik Lundh, um duradouro log de notícias do Python na Web, no endereço *http://www.pythonware.com/daily/*.

Um voluntário, em sistema de rodízio, também monitora a python-list e a resume semanalmente. Patrocinados pelo *Doctor Dobb's Journal*, repositórios de arquivo e informações sobre como se inscrever estão disponíveis no endereço *http://www.ddj.com/topics/pythonurl/*.

O PROCESSO

O assunto sobre como o Python é desenvolvido é fascinante, pois ele é um dos melhores projetos de código-fonte aberto em andamento, mas isso está fora dos objetivos deste livro, de modo geral. Se você estiver interessado em aprender mais (ache você que pode colaborar com o Python ou não), leia algumas das informações no endereço *http://www.python.org/dev* – você encontrará de tudo, desde descrições da cultura de desenvolvedor de Python até detalhes técnicos específicos sobre como colaborar.

Entretanto, como usuário de Python, você pode contribuir no caso (improvável) de encontrar um erro na linguagem. Se você encontrar, deve isolar o código que está causando problemas ou não se comportando minimamente de acordo com a especificação e postá-lo como erro no rastreador de erros do Python. Quando este livro estava sendo produzido, o Python estava usando o rastreador de erros executado pela Sourceforge, embora houvesse boatos sobre mudança para outro lugar. O gerenciador de erros está localizado no endereço *http://www.dourceforge.net/bugs/?group_id=5470* e as instruções sobre como enviar um relatório de erro estão no endereço *http://www.python.org/doc/current/ext/reporting-bugs.html*.

SERVIÇOS E PRODUTOS

Existem centenas de milhares ou, talvez, milhões de pessoas usando Python e milhares de empresas contando com ele. Vários fornecedores comerciais, tanto corporações como consultores individuais, fornecem suporte de vários tipos para ajudar pessoas e empresas a trabalharem com o Python, desde treinamento e ferramentas de desenvolvimento, até suporte por telefone. Um fato não totalmente surpreendente é que o principal trabalho do primeiro autor é ensinar Python para pessoas e empresas de todo o mundo e a empresa do segundo autor fornece ferramentas de desenvolvimento e suporte em nível empresarial para Python. Existem muitos outros fornecedores – consulte seus canais normais para localizar o provedor mais conveniente para suas necessidades.

A ESTRUTURA JURÍDICA: A PYTHON SOFTWARE FOUNDATION

Em seu núcleo, o Python é uma linguagem de programação. Como tecnologia, o Python precisa de um proprietário – uma pessoa ou entidade que possa defini-lo para o mundo, protegê-lo de ataques e sustentar seu crescimento. Embora Guido van Rossum seja reconhecido como o pai do Python, ele não quer ser a única pessoa responsável pela linguagem. As discussões sobre "e se Guido for atropelado por um ônibus" têm sido tratadas, com o passar dos anos, por meio da definição de uma entidade jurídica chamada Python Software Foundation (PSF), para atuar como proprietária legal do Python. Assim, toda propriedade intelectual do Python está sendo atribuída à PSF. A PSF recebeu o status provisório de empresa sem fins lucrativos do IRS dos EUA, tornando assim as doações feitas para a organização dedutíveis do imposto de renda. A PSF é composta por membros individuais (convidados pelo quadro de associados existente pela sua colaboração no Python) e é financiada por patrocinadores corporativos. Informações sobre a PSF estão disponíveis no endereço *http://www.python.org/psf/*. Os dois autores são membros da PSF e o segundo autor é diretor dela. Basicamente, o que isso significa é que a PSF é uma organização que acreditamos ser importante para a saúde do Python a longo prazo.

SOFTWARE

A tarefa deste livro foi apresentar a linguagem Python, incluindo breves visões gerais de alguns dos módulos e bibliotecas mais importantes que o acompanham. Existem inúmeros

outros pacotes de software de apoio, a maioria deles gratuita, na Internet. Nesta seção, fornecemos os indicadores de como é esse panorama de software, quais mapas estão disponíveis para ajudá-lo a encontrar o que está procurando e, finalmente, alguns pacotes de software notáveis que podem tornar a escolha do Python algo de alto valor.

Uma das deficiências do Python tem sido a falta de um repositório único e autorizado de software de outros fornecedores. Embora existam voluntários trabalhando duro para resolver esse problema, o melhor que pudemos fazer, quando este livro estava sendo produzido, foi apontar vários métodos alternativos que podem ser usados para descobrir o que está disponível e onde.

Pesquise na Web

Encontrar coisas na Internet costumava ser difícil. Alguns de nós lembramos dos tempos anteriores à Web, quando contatos verbais e cumprimentos secretos pareciam ser obrigatórios para se encontrar um programa de software em particular. Atualmente, mecanismos de pesquisa como o Google fazem 95% do trabalho difícil. Independente do assunto, é muito provável que as pesquisas no Google forneçam o que você deseja.

Pesquise nos repositórios de arquivo de listas de distribuição

Normalmente, o software que está disponível na Web foi anunciado publicamente ou, no mínimo, discutido em público. Você pode pesquisar as diversas listas de distribuição mencionadas anteriormente, com mecanismos de pesquisa especializados, como a interface Groups do Google (embora isso não cubra todas as listas de distribuição do Python, mas apenas aquelas espelhadas como newsgroups), no endereço *http://search.python.org*, ou os repositórios de listas de distribuição, no endereço *http://www.ASPN.ActiveState.com*.

Procure no Vaults of Parnassus

O "The Vaults of Parnassus" é um catálogo antigo e bem estabelecido de software Python. Ele usa um diretório estilo biblioteca de software Python e de ferramentas relacionadas à linguagem. Os cofres (vaults) estão no endereço: *http://py.vaults.ca/*. Note que os cofres arquivam apenas metadados e encontrar algo neles não garante que as páginas a que se referem ainda existam, nem que as informações são necessariamente atualizadas.

Consulte o Python Package Index (PyPI)

Um novo projeto que, ao contrário de alguns de seus predecessores, parece ter sucesso é chamado PyPI (Python Package Index). Hospedado no endereço *http://www.python.org/pypi*, o protótipo atual permite que as pessoas registrem seus pacotes python manualmente ou, preferivelmente, usando o software de distribuição de pacote `distutils`, que faz parte da biblioteca padrão. Quando este livro estava sendo produzido, havia apenas algumas dezenas de pacotes listados, mas quando você ler esse catálogo, provavelmente será muito maior.

Procure no Python Cookbook

O *Python Cookbook* é um projeto conjunto que combina os esforços da ActiveState, da O'Reilly and Associates e da comunidade Python. A ActiveState hospeda um site da Web (*aspn.ActiveState.com/Python/CookBook*) que permite a qualquer um postar sua receita predileta de código Python e solicitar o retorno dos leitores do site. A O'Reilly publicou um livro com receitas selecionadas, editadas e ampliadas, chamado *Python Cookbook*, co-editado por

Alex Martelli e David Ascher. O livro contém centenas de receitas bem motivadas e explicadas em detalhes, e se tornou popular, mesmo entre os programadores de Python veteranos. O site on-line contém centenas de outras receitas e está constantemente sendo atualizado. Ambos são recursos excelentes para esse tipo de pacote de software menor, os trechos de código e idiomas que definem o falante fluente de uma linguagem.

SOFTWARE POPULAR DE OUTROS FORNECEDORES

Nesta seção, listamos alguns dos complementos de terceiros mais populares para o Python. Alguns são módulos pequenos, embora extremamente úteis, outros são aplicativos completos de alta complexidade interna. Cada um deles é o que consideramos uma boa ferramenta.

As URLs mudam; portanto, não fique desapontado se as URLs que mencionamos não forem mais válidas quando você as digitar. Em vez disso, vá no Google e digite python *nome do pacote* – é muito provável que você a encontre.

Interfaces para Windows e MacOS

Embora cada sistema operacional forneça uma grande variedade de interfaces, o Unix e os sistemas operacionais tipo Unix relacionados tendem a fornecer essa interface por meio de ferramentas de linha de comando e arquivos de propósito especial, e ambos tendem a variar muito entre as versões para possibilitar a existência de interfaces programáticas úteis. O Windows e o Macintosh usam uma estratégia mais voltada à API e, como resultado, tornam o sistema operacional mais naturalmente acessível a partir de uma linguagem de programação como o Python. Existem interfaces Python para praticamente qualquer parte das APIs Windows e Macintosh.

Windows

O Python básico vem com algumas interfaces para interfaces básicas do Windows, como o módulo `os` para as funções básicas do sistema operacional e a API de baixo nível `_regedit` para o Registro do Windows. Entretanto, uma programação séria do Windows exige acesso a muito mais bibliotecas do Windows. A maioria delas são expostas pelo pacote `win32all`, de Mark Hammond. O `win32all` está disponível como um complemento para a distribuição de Python no endereço *http://www.python.org* ou empacotado como parte da ActivePython da ActiveState (*http://www.ActiveState.com/Python*). O `win32all` também inclui um IDE somente para Windows do Python, o Pythonwin.

Nem todas as APIs Windows são expostas pelo pacote `win32all`. Se você quiser usar uma delas, pode utilizar o módulo `ctypes`, de Thomas Heller, que fornece uma interface de função estrangeira do Python para bibliotecas compartilhadas, carregadas dinamicamente. O módulo `ctypes` está descrito a seguir.

Macintosh

A implementação no Macintosh do Python, mantida por Jack Jansen e disponível no endereço *http://www.cwi.nl/~jack/macpython.html*, existia em dois tipos, quando este livro estava sendo produzido. Existe uma nova versão que é executada a partir da linha de comando do Mac OS X, assim como uma versão que é executada no Mac OS 9 ou OS X, embora existam planos para mesclar as duas. A documentação da biblioteca Macintosh, a *Macintosh Library Modules*, faz parte da referência de biblioteca padrão. As versões recentes do Mac OS X já vêm com o Python previamente instalado.

Bibliotecas de propósito especial

Existem alguns domínios de computação que servem como bibliotecas bem-definidas. Normalmente, eles envolvem tipos de dados especializados (por exemplo, arrays de números, datas) ou algoritmos de propósito especial (por exemplo, mecanismos de elementos gráficos tridimensionais). Nesta seção, fazemos um levantamento de algumas dessas bibliotecas mais populares.

Bibliotecas de computação científica

O Python tem seguidores significativos nos campos científico e de engenharia. Sua sintaxe matemática bastante razoável e sua facilidade de extensão o tornam bom para cientistas que precisam combinar sintaxe fácil de usar com a capacidade de anexar mecanismos computacionais de alto poder.

A avó das bibliotecas científicas no Python é chamada de Numeric Python, NumPy ou, abreviadamente, Numeric. A Numeric consiste em um conjunto de módulos de extensão Python e C que fornecem operações de array muito poderosas e de alto desempenho. Com a Numeric, pode-se efetuar operações "em massa", em arrays multidimensionais de números muito grandes, muito mais rápido do que se consegue no Python puro. A Numeric é a solução corrente padrão para executar essas operações em Python, embora muitos estejam examinando sua substituta proposta, a numarray.

Aqui está um exemplo de código NumPy típico, *numpytest.py*, e uma representação dos dados que ele gera:

```
from Numeric import *
coords = arange(-6, 6,.02)              # Cria um intervalo de coordenadas.
xs = sin(coords)                        # Pega o seno de todos os x.
ys = cos(coods)*exp(-coords*coords/18.0) # Pega uma função complexa dos y.
zx = xs * ys[:,NewAxis]                 # Multiplica a linha x com a coluna y.
```

Se você lembrar da matemática, pode descobrir que `xs` é um array de senos dos números entre –6 e 6 e `ys` é um array dos cossenos desses mesmos números, colocados em escala por uma função exponencial centralizada em 0. `zs` é simplesmente o produto externo desses dois arrays de números. Se você estiver curioso quanto a como isso pode parecer, pode converter o array `zs` em uma imagem e obter a imagem mostrada na Figura 29-1.

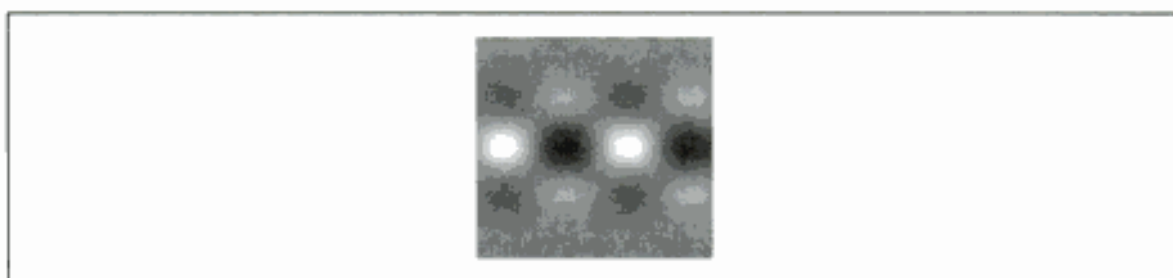



Figura 29-1 Representação gráfica do array zs em numpytest.py.

A NumPy permite manipular arrays muito grandes de forma muito eficiente. O código anterior é executado muito mais rapidamente do que um código comparável usando listas grandes de números e utiliza uma fração da memória. Muitos usuários de Python nunca precisam lidar com esses tipos de problemas, mas muitos cientistas e engenheiros exigem tais recursos diariamente.

A numarray (*http://stsdas.stsci.edu/numarray/*) é uma reimplementação da funcionalidade básica da Numeric, enquanto almeja corrigir alguns dos problemas da original, relacionados

às conversões de tipo, capacidade de extensão, eficiência da memória e outros fatores. É cedo demais para dizer, mas nossas fontes indicam que a numarray está bem posicionada para tomar a fatia de mercado da Numeric, em uma transição correta, progressiva e orquestrada.

Para quem estiver interessado em fazer mais coisas com números, vale a pena examinar o projeto SciPy. O SciPy complementa os módulos de tratamento de array básicos com uma variedade de módulos científicos e de engenharia de alto nível, como solucionadores, otimizadores, rotinas estatísticas e uma ferramenta em linha muito interessante que compila expressões usando C++ de maneira semi-transparente, chamada weave. O SciPy também inclui o software gráfico científico mais "Pythonico", chamado Chaco. Informações sobre o SciPy podem ser encontradas em seu site da Web, no endereço *http://www.scipy.org*.

Finalmente, existem outras bibliotecas científicas para o Python, não listadas anteriormente – algumas para tratar com bibliotecas Fortran, outras para realizar processamento especializado nos campos da química, genética etc. Use os recursos mencionados para procurar software em seu campo em particular.

Interfaces de bancos de dados relacionais

Os bancos de dados relacionais são um modo muito comum de armazenar e manipular dados, embora, conforme vimos neste livro, existam outras maneiras mais simples de armazenar, pelo menos para pequenos volumes de dados (como usar `pickle` ou os módulos relacionados `marshal` e `shelve` – consulte sua documentação para ver os detalhes). Todas as linguagens de script populares têm interfaces para mecanismos de banco de dados comuns e o Python não é exceção. Existe até uma interface padrão que permite aos programadores de Python se comunicarem com uma variedade de bancos de dados de maneira independente do banco, chamada DB-API. Os detalhes sobre a DB-API, incluindo referências para módulos correntes, estão disponíveis no endereço *http://www.python.org;topics/database/*. Você verá que existem interfaces para Oracle, Postgres, MySQL, SAP DB, Informix e muitos outros.

Existem outros bancos de dados menos padronizados que vale a pena conhecer, no caso de você não ter uma razão particular para usar um servidor de SQL. Os dois mais comumente usados são o Gadfly e o MetaKit. O Gadfly (*http://gadfly.sourceforge.net/gadfly.html*) é um servidor de SQL escrito inteiramente em Python (com algumas extensões da linguagem C) que fornece uma solução de SQL muito simples para as pessoas que têm bancos de dados relativamente pequenos (o Gadfly armazena seus dados na memória) e não exigem recursos multiusuário, mas ainda querem suporte transacional. O MetaKit (*http://www.equi4.com/metakit/python.html*) é um mecanismo de banco de dados em C++ com vínculos para o Python muito bons. O MetaKit tem uma visão do mundo mais orientada para registros e é notadamente rápido em um conjunto de operações notável. Uma terceira opção, mais recente, entre o Gadfly e os bancos de dados SQL completos é o PySQLite, uma interface para o popular mecanismo SQL incorporado SQLite.

Uma última possibilidade que vale a pena mencionar aqui é o conjunto de ferramentas da eGenix.com. Elas incluem a mxODBC (uma interface para a camada de banco de dados ODBC), a mxDateTime (uma biblioteca de data/hora de alto poder) e outros complementos. Elas estão disponíveis no endereço *http://www.egenix.com*).

Bibliotecas gráficas

Existem vários tipos de pacotes que podem ajudar a produzir saída gráfica de vários tipos, desde ferramentas de processamento de imagens, modelos para jogos, até kits de ferramentas de visualização.

Python Imaging Library (PIL). A Python Imaging Library é um modelo amplo, escrito por Frederik Lundh, para criar, manipular, converter e salvar imagens de mapa de bits em uma variedade de formatos, como GIF, JPEG e PNG. Ele tem interfaces para Tk e Pythonwin, de modo que se pode usar elementos de janela Tk ou código Pythonwin para exibir as imagens geradas pelo PIL. Como alternativa, as imagens podem ser salvas no disco em uma variedade de formatos. O PIL está localizado no endereço *http://www.pythonware.com*.

PyGame e PyOpenGL. Duas das escolhas mais populares para trabalho com elementos gráficos são o PyGame, para software bidimensional e de jogos, e o PyOpenGL, para elementos gráficos tridimensionais, especialmente na área da visualização científica. O PyGame (*http://pygame.org*) é um conjunto de módulos Python escritos em cima da biblioteca gratuita e independente de plataforma SDL. Ele tem sido usado para uma variedade de software de jogos e outros, e fornece interfaces excelentes para a tela, mouse, teclado e saída de som. As pessoas têm escrito alguns jogos interessantes com PyGame, assim como utilizado em aplicações que não são jogos, com pesada ênfase em animações tipo jogo, controles etc. Um exemplo simples de programação com PyGame é o trecho de código a seguir, que toca os primeiros cinco segundos da primeira trilha do CD player que encontra e depois o ejeta.

```
import pygame
pygame.cdrom.init()
cd_object = pygame.cdrom.CD(pygame.cdrom.get_count()-1)
cd_object.init()
if cd_object.get_track_audio(i):
    audio_track_found = 1
    cd_object.play(i)
    pygame.time.delay(5000)
cd_object.eject()
cd_object.quit()
```

O PyOpenGL é uma ferramenta muito diferente – trata-se de um envoltório em torno do OpenGL, o padrão para elementos gráficos bi e tridimensionais de alto desempenho e independente de plataforma. O PyOpenGL, disponível no endereço *http://pyopengl.sourceforge.net*, fornece interfaces em nível de Python para um número muito grande de APIs OpenGL, assim como bibliotecas relacionadas, como GLU, GLUT e muito mais. O PyOpenGL, especialmente quando combinado com arrays da Numeric Python e usado com uma placa gráfica que forneça aceleração de hardware do OpenGL, pode produzir elementos gráficos espantosamente rápidos. Qualquer um que esteja procurando fazer visualização científica ou de engenharia deve explorar essa opção.

Interfaces para kits de ferramentas de GUI

Trabalhar com um kit de ferramentas de GUI é um requisito para muitas pessoas que escrevem software de usuário final. A decisão sobre qual kit de ferramentas de GUI usar normalmente é muito complexa, envolvendo fatores como portabilidade, desempenho, aparência e comportamento, licença e custo, modelos de segmentação etc. Basta dizer que o Python tem interfaces para todos os kits de ferramentas de GUI nas principais plataformas.

Existem vários kits de ferramentas de plataforma populares com interfaces para Python:

- O Tk tem uma interface que acompanha o Python padrão, chamada Tkinter. Embora o conjunto de elementos de janela do Tk não seja tão rico quanto os outros, existem maneiras de estendê-lo usando ferramentas megawidget, como o Pmw (*http://pmw.sf.net*).
- wxPython, a interface para o kit de ferramentas de GUI wxWindows, tornou-se muito popular recentemente (consulte o endereço *http://www.wxPython.org* para ver os detalhes).

a possibilidade de fazer código Python ser executado com mais rapidez do que código em C, que é considerado como referência para linguagens interpretadas. Quem estiver curioso pode ler sobre o projeto Psyco corrente no endereço *http://psyco.sourceforge.net/*, assim como um projeto relacionado, pypy, cujo objetivo é escrever uma nova implementação de Python em Python e construir tecnologia estilo Psyco para compilá-lo (veja no endereço *http://codespeak.net/pypy/*).

ctypes

A última entrada neste conjunto de ferramentas é a `ctypes`. A ctypes usa uma estratégia de interface completamente diferente – ao contrário de SWIG, Boost::Python ou do sistema de módulo de extensão C padrão, a ctypes é projetada para chamar diretamente do Python em bibliotecas compartilhadas. Usando truques específicos da plataforma (a ctypes funciona atualmente apenas em Windows, Linux e OS/X, embora mais plataformas possam ser adicionadas), a ctypes permite carregar bibliotecas compartilhadas arbitrárias e chamar funções arbitrárias, contanto que você conheça a assinatura dessas funções:

```
>>> from ctypes import cdll, c_double
>>> printf = cdll.msvcrt.printf
>>> printf("An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
```

Há muito mais na `ctypes` que torna essa estratégia realista para uma ampla variedade de aplicações. Consulte o endereço *http://starship.python.net/crew/theller/ctypes.html* para ver os detalhes.

Pequenas preciosidades

Nesta seção, listaremos alguns módulos que, embora não sejam necessariamente complexos demais nem sejam trabalhos maiores, fornecem excelentes soluções para problemas comuns.

O módulo `platform` (parte da biblioteca padrão no Python 2.3, mas disponível para versões anteriores, no endereço *http://www.egenix.com/files/python/platform.py*) permite descobrir muito mais variações sutis do que sua plataforma corrente, comparado à `sys.platform`. Em vez de apenas dizer win32, ele permite que você saiba que está executando em, por exemplo, `"Windows-2000-5.0.2195-SP3"`. Às vezes, esse tipo de informação é fundamental e é difícil identificar sem um código como o que há em *platform.py*.

O *turtle.py* (que faz parte da distribuição padrão, mas é desconhecido de muitos) fornece um sistema simples estilo LOGO para desenho interativo e programadores iniciantes. Construído no Tkinter, ele fornece um bom primeiro passo para ensinar programação.

O `go`, de Trent Mick (no endereço *http://starship.python.net/crew/tmick*) é uma ferramenta de linha de comando simples que torna trivial construir marcadores de diretórios utilizados freqüentemente. Em vez de ter de lembrar longos nomes de caminho, você pode dizer ao `go` quais diretórios utiliza freqüentemente e, então, usar comandos de duas palavras para chegar até lá (`go home`, `go python23` etc.).

Ferramentas de empacotamento

Os desenvolvedores de Python que queiram compartilhar seu código com outros desenvolvedores devem aprender a usar o pacote `distutils`, o qual permite às pessoas construir distribuições de seus programas que são fáceis para outros desenvolvedores instalarem. O pacote

APÊNDICE A
Instalação e Configuração

Este apêndice fornece detalhes adicionais sobre instalação e configuração, como um recurso para os iniciantes nesses assuntos.

INSTALANDO O INTERPRETADOR DO PYTHON

Como você precisa do interpretador do Python para executar scripts, o primeiro passo para usar a linguagem normalmente é instalá-la. A não ser que o Python já esteja disponível em sua máquina, você precisará buscá-lo, instalá-lo e possivelmente configurá-lo em seu computador. Você só precisa fazer isso uma vez em cada máquina e talvez não precise fazer absolutamente nada, se for executar um binário congelado.

Onde obter o Python

Primeiramente, antes de fazer qualquer coisa, certifique-se se você já não tem um Python recente em sua máquina. Por exemplo, se você estiver trabalhando no Linux ou em alguns sistemas Unix, o Python provavelmente já está instalado. Digite **python** em um prompt de shell e veja o que acontece. Como alternativa, tente pesquisar os lugares usuais (*/usr/bin, /usr/local/bin* etc.). No Windows, verifique se existe uma entrada para o Python no menu de programas que você encontra no botão Iniciar, na parte inferior esquerda da tela. Certifique-se de que o Python encontrado seja a versão 2.2 ou posterior; você precisará disso para executar alguns dos exemplos deste livro.

Se não houver nenhum Python, você mesmo precisará instalá-lo. Você sempre pode buscar a versão padrão melhor e mais recente do Python, no endereço *http://python.org*, o site da Web oficial da linguagem. Procure o link Downloads nessa página e selecione uma versão para a plataforma em que você vai trabalhar. Lá, você encontrará executáveis do Python previamente compilados (desempacote e execute), executáveis de instalação automática para Windows (dê um clique para instalar), RPMs para Linux (desempacote com rpm), a distribuição em código-fonte completa (compile em sua máquina para gerar um interpretador) e muito mais. Para algumas plataformas, como PalmOS e PocketPC, o site da Web do Python tem links para uma página em outro local onde essas versões são mantidas.

Você também pode encontrar o Python em CD-ROMs fornecidos com distribuições de Linux incluídas em alguns produtos e sistemas de computador, vendidas por distribuidores comerciais, como o *Dr. Dobb's Journal*, e contidas em outros livros de Python. Elas tendem a estar atrasadas em relação à versão corrente, mas normalmente, não muito.

Além disso, uma empresa chamada ActiveState também distribui o Python como parte de seu pacote *ActivePython*. Esse pacote combina o CPython padrão com extensões para desenvolvimento no Windows, um IDE chamado PythonWin (descrito no Capítulo 3) e outras extensões comumente usadas. Consulte o site na Web da ActiveState, no endereço *www.activestate.com*, para ver mais detalhes sobre o pacote ActivePython.

Finalmente, se você estiver interessado em implementações do Python alternativas, procure o Jython no endereço *www.jython.org* e o Python.NET no site da Web da ActiveState; a instalação desses sistemas está fora dos objetivos deste livro.

Etapas de instalação

Uma vez que você tenha o Python, ele deve ser instalado. As etapas de instalação são muito específicas da plataforma, mas aqui estão algumas indicações para as principais plataformas Python:

Windows

No Windows, o Python vem como um arquivo de programa executável de instalação automática – basta dar um clique duplo em seu ícone de arquivo e responder Sim ou Avançar em cada prompt, para fazer uma instalação padrão. A instalação padrão inclui o conjunto de documentação do Python e suporte para GUIs do Tkinter, bancos de dados comuns, a GUI de desenvolvimento IDLE e outras coisas que descreveremos posteriormente. Após a instalação, o Python aparece no menu de programas de seu botão Iniciar.

A Figura A-1 mostra onde o Python aparece no botão Iniciar em uma máquina Windows XP, após a instalação. O menu do Python tem cinco entradas que fornecem rápido acesso às tarefas comuns: iniciar a interface com o usuário IDLE, ler documentação de módulo, iniciar uma sessão interativa, ler os manuais padrão do Python em um navegador da Web e desinstalar. A maioria dessas ações envolve conceitos que exploramos em detalhes neste livro.

Quando instalado no Windows, o Python também se registra automaticamente para ser o programa que abre arquivos Python quando seus ícones recebem um clique de mouse – uma técnica de ativação de programa descrita no Capítulo 3. Também é possível construir o Python a partir de seu código-fonte no Windows, mas isso não é comum.

Linux

No Linux, o Python está disponível como um ou mais arquivos RPM, os quais você desempacota da maneira usual. Consulte a página de manual de seu RPM para ver os detalhes. Dependendo de quais RPMs você busca por download, pode haver um para o próprio Python e outro que adiciona suporte para GUIs do Tkinter e para o ambiente IDLE. Como o Linux é um sistema tipo Unix, o próximo parágrafo também se aplica a ele.

Unix

Nos sistemas Unix, o Python normalmente é compilado a partir de sua distribuição em código-fonte em C completo. Isso normalmente exige apenas desempacotar o arquivo e executar comandos `config` e `make` simples; o Python configura seu próprio procedimento de construção, automaticamente, de acordo com o sistema em que está sendo compilado. Entretanto, leia o arquivo *README* do pacote para ver mais detalhes sobre esse processo. Como o Python é código-fonte aberto, seu código pode ser usado e distribuído gratuitamente.

Em outras plataformas, esses detalhes podem diferir bastante; instalar o portão ports "Pippy" do Python para o PalmOS, por exemplo, exige uma operação de sincronismo automático com seu PDA, e o Python para o PDA baseado em Linux Sharp Zaurus vem como um ou mais arquivos *.ipk*, os quais você simplesmente executa para instalar. Contudo, como os procedimentos de instalação adicionais para as formas de arquivo executável e de código-fonte são bem documentados, não mostraremos mais detalhes aqui.

Etapas de configuração

Após ter instalado o Python, você também pode configurar ajustes de sistema que afetam o modo como a linguagem executa seu código. Se você está apenas começando a usar a linguagem, provavelmente pode pular esta seção completamente; normalmente, não há necessidade de fazer nenhum ajuste de sistema para programas básicos.

Contudo, de modo geral, as partes relativas ao comportamento do interpretador do Python podem ser configurados com ajustes de *variável de ambiente* e com *opções de linha de comando*. Nesta seção, examinamos brevemente as variáveis de ambiente. As opções de linha de comando do Python – as palavras listadas quando você executa um programa em Python a partir de um prompt de sistema – são usadas mais raramente e têm funções muito especializadas; consulte outras fontes de documentação para ver os detalhes das opções de linha de comando do Python.

Variáveis de ambiente do Python

As variáveis de ambiente – conhecidas por alguns como variáveis de shell ou variáveis DOS – ficam fora do Python e, assim, podem ser usadas para personalizar o comportamento do interpretador cada vez que ele é executado em determinado computador. O Python reconhece diversas configurações de variável de ambiente, mas apenas algumas são usadas com freqüência suficiente para merecer uma explicação aqui. A Tabela A-1 resume as principais configurações de variável de ambiente relacionadas ao Python.

Tabela A-1 Variáveis de ambiente importantes

Variável	Função
PATH (ou path)	Caminho de pesquisa do shell do sistema (para localizar "python")
PYTHONPATH	Caminho de pesquisa de módulo do Python (para importações)
PYTHONSTARTUP	Caminho para o arquivo de inicialização interativo do Python
TCL_LIBRARY, TK_LIBRARY	Variáveis de extensão de GUI (Tkinter)

Essas variáveis são simples de usar, mas aqui estão algumas indicações:

- A configuração de `PATH` lista um conjunto de diretórios que o sistema operacional pesquisa para encontrar programas executáveis. Normalmente, ela deve incluir o diretório onde está seu interpretador do Python (o programa python no Unix ou o arquivo *python.exe*, no Windows). Você não precisa configurar essa variável, caso queira trabalhar no diretório onde o Python reside ou digitar o caminho completo para o Python em linhas de comando. No Windows, por exemplo, a variável `PATH` será irrelevante se você executar `cd C:\Python22`, antes de executar qualquer código (para mudar para o diretório onde o Python está), ou sempre digitar **`C:\Python22\python`**, em vez de digitar apenas `python` (para fornecer um caminho completo). Note também que as configurações de PATH ser-

vem principalmente para executar programas a partir de linhas de comando; normalmente elas são irrelevantes para executar com cliques de mouse em ícones e com IDEs.

- A configuração de `PYTHONPATH` tem uma função semelhante à de `PATH`: o interpretador do Python consulta essa variável para localizar arquivos de módulo, quando você os importa em um programa. (Falamos sobre o caminho de pesquisa de módulo no Capítulo 15.) Se for usada, essa variável é configurada com uma lista de nomes de diretório dependente da plataforma, lista essa separada por dois-pontos no Unix e por pontos-e-vírgulas no Windows. Essa lista normalmente inclui apenas os seus diretórios de código-fonte. Você também não precisa configurar essa variável, a não ser que vá realizar importações entre diretórios – como o Python sempre pesquisa automaticamente o diretório de base do arquivo de nível superior do programa, essa configuração só é exigida se um módulo precisa importar outro módulo residente em um diretório diferente. Conforme mencionado no Capítulo 15, os arquivos *.pth* são uma alternativa recente à variável `PYTHONPATH`.
- Se a variável `PYTHONSTARTUP` for configurada com o nome de caminho de um arquivo de código Python, a linguagem executará o código do arquivo automaticamente, quando você iniciar o interpretador interativo, como se tivesse sido digitado na linha de comando interativa. Isso é usado raramente, mas é uma maneira interessante de garantir que você sempre carregue utilitários ao trabalhar interativamente; isso economiza uma importação.
- Se você quiser usar o kit de ferramentas de GUI do Tkinter, talvez tenha que configurar as duas variáveis de GUI da Tabela A-1 com o nome dos diretórios de biblioteca de código-fonte dos sistemas Tcl e Tk (de forma muito parecida com a variável `PYTHONPATH`). Entretanto, essas configurações não são exigidas em sistemas Windows (onde o suporte para Tkinter é instalado junto com a Python) e normalmente não são exigidas em outros lugares, se o Tcl e o Tk residirem em diretórios padrão.

Note que, como essas configurações de ambiente (assim como os arquivos *.pth*) são externas ao Python em si, o *momento* em que você as configura normalmente é irrelevante. Elas podem ser configuradas antes ou depois que o Python for instalado – contanto que sejam configuradas da maneira exigida, antes que o Python seja realmente *executado*.

Como ajustar opções de configuração

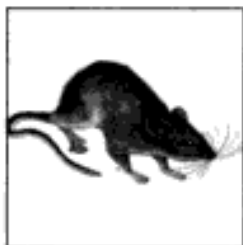
A maneira de configurar essas variáveis, e para o que configurá-las, depende do tipo de computador em que você vai trabalhar. E, novamente, lembre-se de que você não precisa necessariamente configurá-las imediatamente; particularmente ao se trabalhar sob o IDLE (descrito no Capítulo 3), a configuração antecipada não é exigida.

Mas, por ilustração, suponha que você tenha arquivos de módulo úteis de modo geral em diretórios chamados *utilities* e *package1*, em algum lugar em sua máquina, e queira importar esses módulos a partir de arquivos localizados em qualquer outro diretório. Para carregar um arquivo chamado *spam.py* a partir do diretório *utilities*, você quer escrever:

```
import spam
```

a partir de outro arquivo localizado em qualquer lugar em seu computador. Para fazer isso funcionar, você terá que configurar seu caminho de pesquisa de módulo de uma maneira ou de outra, para incluir o diretório que contém `spam.py`. Aqui estão algumas dicas sobre esse processo.

Os nomes de diretório em arquivos de caminho podem ser absolutos ou relativos ao diretório que contém o arquivo de caminho; vários arquivos *.pth* podem ser usados (todos os seus diretórios são adicionados) e os arquivos *.pth* podem aparecer em diversos diretórios verificados automaticamente, que sejam específicos da plataforma e da versão. Por exemplo, a versão 2.2 normalmente procura arquivos de caminho em *C:\Python22* e em *C:\Python22\Lib\site-packages* no Windows, e em */usr/local/bin/python2.2/site-packages* e em */usr/local/lib/site-python* no Unix e no Linux.

Contudo, como essas configurações freqüentemente são opcionais e como este não é um livro sobre shells de sistema operacional, vamos deixar os detalhes para outra documentação. E se você tiver problemas para descobrir quais devem ser as suas configurações, peça ajuda para seu administrador de sistema ou outro especialista local.

APÊNDICE B
Soluções dos Exercícios

PARTE I, COMEÇANDO

1. *Interação.* Supondo que o Python esteja configurado corretamente, a interação deve ser parecida com a seguinte. Você pode executar isso como quiser: no IDLE, a partir de um prompt de shell etc.:

```
% python
...linhas de informação de direitos de cópia...
>>> "Hello World!"
'Hello World!"
>>>            # Pressione Ctrl-D ou Ctrl-Z para sair ou feche a janela
```

2. *Programas.* Seu arquivo de código (isto é, módulo) *module1.py* e as interações do shell devem ser como segue:

```
print 'Hello module world!'

% python module1.py
Hello module world!
```

Novamente, sinta-se à vontade para executar isso de outras maneiras – dando um clique em seu ícone, por meio da opção de menu Edit/RunScript do IDLE etc.

3. *Módulos.* A listagem de iteração a seguir ilustra a execução de um arquivo de módulo por meio de sua importação.

```
% python
>>> import module1
Hello module world!
>>>
```

Lembre-se de que você precisa recarregar o módulo para executar novamente, sem parar e reiniciar o interpretador. As perguntas sobre mover o arquivo para um diretório diferente e importá-lo novamente são complicadas: se o Python gera um arquivo *module1.pyc* no diretório original, ele usa isso quando você importa o módulo, mesmo que o arquivo de código-fonte (*.py*) tenha mudado para um diretório que não está no caminho de pes-

método append que altera o array no local, anexando outra referência de objeto. Aqui, a chamada de append adiciona uma referência na frente e no final de L, o que leva ao ciclo ilustrado na Figura B-1. Acredite se quiser, às vezes as estruturas de dados cíclicas podem ser úteis (mas não quando impressas!).

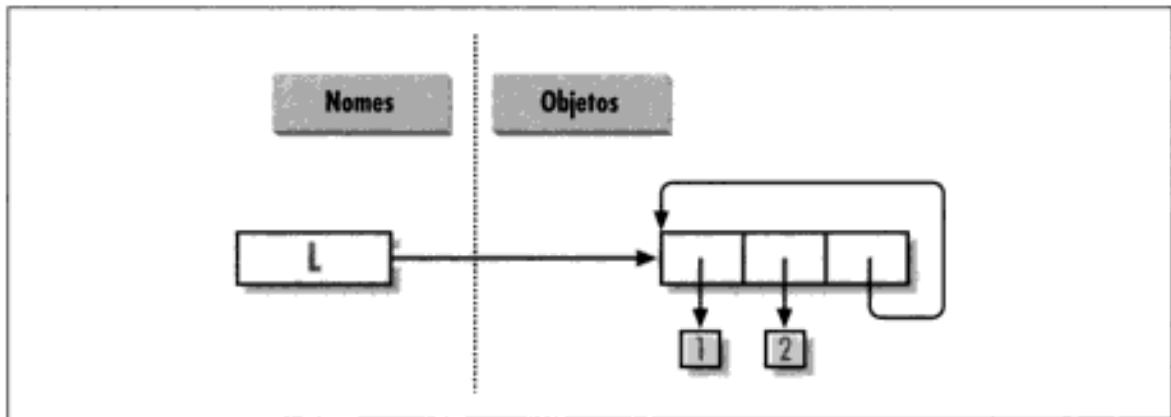



Figura B-1 Uma lista cíclica.

PARTE II, TIPOS E OPERAÇÕES

1. *Os fundamentos.* Aqui estão os tipos de resultados que você deve obter, junto com alguns comentários sobre seus significados. Note que ; é usado em alguns deles para espremer mais de uma instrução em uma única linha. O ; é um separador de instrução.

```
Números

>>> 2 ** 16                 # 2 elevado à potência 16
65536
>>> 2 / 5, 2 / 5.0          # Inteiro / trunca, float / não
(0, 0.40000000000000002)

Strings

>>> "spam" + "eggs"         # Concatenação
'spameggs'
>>> S = "ham"
>>> "eggs" + S
'eggs ham'
>>> S * 5                   # Repetição
'hamhamhamhamham'
>>> S[:0]                   # Um fracionamento vazio na frente--[0:0]
''
>>> "green %s and %s" % ("eggs", S)   # Formatação
'green eggs and ham'

Tuplas

>>> ('x',)[0]               # Indexação de uma tupla de um item
'x'
>>> ('x', 'y')[1]           # Indexação de uma tupla de 2 itens
'y'

Listas
```

```
>>> L = [1,2,3] + [4,5,6]      # Operações de lista
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> ([1,2,3]+[4,5,6])[2:4]
[3, 4]
>>> [L[2], L[3]]               # Busca a partir de deslocamentos; armazena em
                                 uma lista
[3, 4]
>>> L.reverse(); L             # Método: inverte lista no local
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L                # Método: ordena lista no local
[1, 2, 3, 4, 5, 6]
>>> L.index(4)                 # Método: deslocamento dos 4 primeiros (busca)
3

Dicionários

>>> {'a':1, 'b':2}['b']        # Indexa um dicionário pela chave.
2
>>> D = {'x':1, 'y':2, 'z':3}
>>> D['w'] = 0                 # Cria uma nova entrada.
>>> D['x'] + D['w']
1
>>> D[(1,2,3] = 4              # Uma tupla usada como chave (imutável)
>>> D
{'w' : 0, 'z' : 3, 'y' : 2, (1, 2, 3): 4, 'x' : 1}
>>> D.keys(), D.values(), D.has_key((1,2,3))      # Métodos
(['w', 'z', 'y', (1, 2, 3), 'x'], [0, 3, 2, 4, 1], 1)

Vazios

>>> [[]], ["",[],(),{},None]         # Muitos nadas: objetos vazios
([[]], ['',[],(),{},None])
```

2. *Indexação e fracionamento.* Indexar além dos limites (por exemplo, `L[4]`) gera um erro; o Python sempre faz verificações para garantir que todos os deslocamentos estejam dentro dos limites de uma seqüência.

   Por outro lado, fracionar além dos limites (por exemplo, `L[-1000:100]` funciona, pois o Python calcula os fracionamentos fora dos limites de modo que eles sempre se ajustem (eles são configurados como zero e com o comprimento da seqüência, se for exigido).

   Extrair o inverso de uma seqüência – com o limite inferior maior do que o superior (por exemplo, `L[3:1]`) – realmente não funciona. Você recebe de volta um fracionamento vazio (`[]`), pois o Python calcula os limites do fracionamento para garantir que o limite inferior seja sempre menor ou igual ao superior (por exemplo, `L[3:1]` é calculado como `L[3:3]`, o ponto de inserção vazio no deslocamento `3`). Os fracionamentos do Python são sempre extraídos da esquerda para a direita, mesmo que você use índices negativos (eles são primeiramente convertidos em índices positivos, adicionando o comprimento). Note que os fracionamentos de três limites do Python 2.3 modificam bastante esse comportamento: `L[3:1:-1]` extrai da direita para a esquerda.

```
>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
```

```
>>> {} + {}
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: bad operand type(s) for +
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: attribute-less object
>>>
>>> {}.keys()
[]
>>> [].keys()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: keys
>>>
>>> [][:]
[]
>>> ""[:]
''
```

8. *Indexação de string*. Como as strings são coleções de strings de um caractere, sempre que você indexa uma string recebe uma string de volta, a qual pode ser indexada novamente. `S[0][0][0][0][0]` apenas fica indexando o primeiro caractere repetidamente. Geralmente, isso não funciona para listas (as listas podem conter objetos arbitrários), a menos que a lista contenha strings.

```
>>> S = "spam"
>>> S[0][0][0][0][0]
's'
>>> L = ['s', 'p']
>>> L[0][0][0]
's'
```

9. *Tipos imutáveis*. Uma das duas soluções a seguir funciona. A atribuição de índice não funciona, pois as strings são imutáveis.

```
>>> S = "spam"
>>> S = S[0] + 'l' + S[2:]
>>> S
'slam'
>>> S = S[0] + 'l' + S[2] + S[3]
>>> S
'slam'
```

10. *Aninhamento*. Aqui está um exemplo:

```
>>> me = {'name' :('mark', 'e', 'lutz'), 'age':'?', 'job':'engineer'}
>>> me ['job']
'engineer'
>>> me['name'][2]
'lutz'
```

11. *Arquivos*. Aqui está uma maneira de criar e ler um arquivo de texto no Python (`ls` é um comando do Unix; use `dir` no Windows):

```
#File: maker.py
file = open('myfile.txt', 'w')
file.write('Hello file world\n')  # Ou: open().write()
file.close()                      # close nem sempre é necessário

#File: reader.py
file = open('myfile.txt', 'r')
print file.read()                 # Ou print open().read()

% python maker.py
% python reader.py
Hello file world!

% ls -l myfile.txt
-rwxrwxrwa  1 0       0         19 Apr 13 16:33 myfile.txt
```

12. *A função dir revisitada.* Aqui está o que você obtém para listas; os dicionários fazem o mesmo (mas com nomes de método diferentes). Note que o resultado de `dir` é expandido no Python 2.2 – você verá um grande conjunto de nomes com sublinhado adicionais na Parte IV. O atributo `__methods__` também desapareceu na versão 2.2, pois não foi implementado consistentemente – em vez disso, use `dir` para buscar listas de atributos:

```
>>> [].__methods__
['append', 'count', 'index', 'insert', remove', 'reverse', 'sort',...]
>>> dir([])
['append', 'count', 'index', 'insert', remove', 'reverse', 'sort',...]
```

PARTE III, INSTRUÇÕES E SINTAXE

1. *Desenvolvimento de loops básicos.* À medida que você trabalhar neste exercício, acabará com um código semelhante ao seguinte:

```
>>> S = 'spam'
>>> for c in S:
...     print ord(c)
...
115
112
97
109
>>> x = 0
>>> for c in S: x = x + ord(c)   # Ou: x += ord(c)
...
>>> x
433

>>> x = []
>>> for c in S: x.append(ord(c))
...
>>> x
[115, 112, 97, 109]

>>> map(ord, S)
[115, 112, 97, 109]
```

2. *Caracteres de barra invertida.* O exemplo imprime o caractere de bip (`\a`) 50 vezes; supondo que sua máquina possa tratar disso, e quando executado fora do IDLE, você pode obter uma série de bips (ou um tom longo, se sua máquina for suficientemente rápida). Nós avisamos!

3. *Ordenando dicionários.* Aqui está uma maneira de resolver esse exercício (veja o Capítulo 6, se isso não fizer sentido). Lembre-se de que você precisa dividir as chamadas de `keys` e de `sort` dessa forma, porque `sort` retorna `None`. No Python 2.2, você pode iterar pelas chaves de dicionário diretamente, sem chamar `keys` (por exemplo, `for key in D:`), mas a lista de chaves não será ordenada como esta, por meio deste código:

```
>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = D.keys()
>>> key.sort()
>>> for key in keys:
...     print key, '=>', D[key]
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
g => 7
```

4. *Alternativas de lógica de programa.* Aqui está um exemplo de código para as soluções. Seus resultados podem variar um pouco; este exercício é projetado principalmente para você mexer com alternativas de código; portanto, tudo que é razoável merece crédito:

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

i = 0
while i < len(L):
    if 2 ** X == L[i]:
        print 'at index', i
        break
    i = i+1
else:
    print X, 'not found'

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

for p in L:
    if (2 ** X), == p:
        print (2 ** X), 'was found at', L.index(p)
        break
else:
    print X, 'not found'

L = [1, 2, 4, 8, 16, 32, 64]
X = 5
```

```
if (2 ** X) in L:
    print (2 ** X), 'was found at', L.index(2 ** X)
else:
    print X, 'not found'

X = 5
L = []
for i in range(7): L.append(2 ** i)
print L

if (2 ** X) in L:
    print (2 ** X), 'was found at', L.index(2 ** X)
else:
    print X, 'not found'

X = 5
L = map(lambda x: 2**x, range(7))
print L

if (2 ** X) in L:
    print (2 ** X), 'was found at', L.index(2 ** X)
else:
    print X, 'not found'
```

PARTE IV, FUNÇÕES

1. *Os fundamentos.* Não há muito a fazer neste exercício, mas note que usar `print` (e, portanto, sua função) é tecnicamente uma operação *polimórfica*, a qual faz a coisa certa para cada tipo de objeto:

```
% python
>>> def func(x): print x
...
>>> func("spam")
spam
>>> func(42)
42
>>> func([1, 2, 3])
[1, 2, 3]
>>> func({'food': 'spam'})
{'food', 'spam'}
```

2. *Argumentos.* Aqui está um exemplo de solução. Lembre-se de que você precisa usar `print` para ver resultados nas chamadas de teste, pois um arquivo não é o mesmo que código digitado interativamente; o Python normalmente não ecoa os resultados de instruções de expressão em arquivos.

```
def adder(x, y):
    return x + y

print adder(2, 3)
print adder('spam', 'eggs')
print adder(['a', 'b'], ['c', 'd'])

% python mod.py
5
```

```
spameggs
['a', 'b', 'c', 'd']
```

3. *varargs*. Duas funções de adição alternativas aparecem no arquivo a seguir, *adders.py*. A parte difícil aqui é descobrir como se faz para inicializar um acumulador com um valor vazio de qualquer tipo que seja passado. A primeira solução usa teste de tipo manual para procurar um inteiro e, caso contrário, um fracionamento vazio do primeiro argumento (supostamente uma seqüência). A segunda solução usa o primeiro argumento para inicializar e percorrer os itens 2 e além, de forma muito parecida com uma das variantes da função `min`, mostradas no Capítulo 13.

   A segunda solução é melhor. Ambas presumem que todos os argumentos têm o mesmo tipo e nenhuma delas funciona em dicionários; conforme vimos na Parte II, o operador + não funciona em tipos misturados nem em dicionários. Poderíamos acrescentar um teste de tipo e um código especial para somar dicionários também, mas isso é crédito extra.

```
def adder1(*args):
    print 'adder1',
    if type(args[0]) == type(0):    # Inteiro?
        sum = 0                     # Inicializa com zero.
    else:                           # senão, com seqüência:
        sum = args[0][:0]           # Usa fracionamento vazio de arg1.
    for arg in args:
        sum = sum + arg
    return sum

def adder2(*args):
    print 'adder2',
    sum = args[0]                   # inicializa como arg1.
    for next in args[1:]:
        sum = sum + next            # soma itens 2..N.
    return sum

for func in (adder1, adder2):
    print func(2, 3, 4)
    print func('spam', 'eggs', 'toast')
    print func(['a', 'b'], ['c', 'd'], ['e', 'f'])

% python adders.py
adder1 9
adder1 spameggstoast
adder1 ['a', 'b', 'c', 'd', 'e', 'f']
adder2 9
adder2 spameggstoast
adder2 ['a', 'b', 'c', 'd', 'e', 'f']
```

4. *Palavras-chave*. Aqui está uma solução para a primeira parte do exercício (arquivo *mod.py*). Para iterar pelos argumentos de palavra-chave, use uma forma `**args` no cabeçalho da função e um loop como este: `for x in args.keys(): use args[x]`.

```
def adder(good=1, bad=2, ugly=3):
    return good + bad + ugly

print adder()
print adder(5)
print adder(5, 6)
```

```
def f4(a, *b, **c): print a, b, c      # Modos mistos

def f5(a, b=2, c=3): print a, b, c     # Padrões

def f6(a, b=2, *c): print a, b, c      # Padrões e varargs posicionais

% python
>>> f1(1, 2)                 # Corresponde pela posição (a ordem importa)
1 2
>>> f1(b=2, a=1)             # Corresponde pelo nome (a ordem não importa)
1 2

>>> f2(1, 2, 3)              # Posicionais extras coletados em uma tupla
1 (2, 3)

>>> f3(1, x=2, y=3)          # Palavras-chave extras coletadas em um dicionário
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, x=2, y=3)    # Extras de dois tipos
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                    # Ambos padrões inseridos.
1 2 3
>>> f5(1, 4)                 # Apenas um padrão usado
1 4 3

>>> f6(1)                    # Um argumento: corresponde a "a"
1 2 ()
>>> f6(1, 3, 4)              # Posicional extra coletado
1 3 (4,)
```

8. *Números primos revisitados.* A seguir está o exemplo dos números primos dentro de uma função e em um módulo (arquivo *primes.py*) para que possa ser executado várias vezes. Adicionamos um teste `if` para capturar números negativos, 0 e 1. Também mudamos / para // para tornar isso imune às alterações na divisão "real" / do Python 3.0 que estudamos no Capítulo 4 e para suportar números de ponto flutuante. O operador // funciona nos esquemas de divisão atual e futuro, mas o futuro operador / falha (retire o comentário de `from` e mude // para /, para ver as diferenças nas versões 2.2 e 3.0).

```
#from __future__ import division

def prime(y):
    if y <= 1:                          # Para algum y > 1
        print y, 'not prime'
    else:
        x = y // 2                      # O futuro / falha
        while x > 1:
            if y % x == 0:              # Nenhum resto?
                print y, 'has factor', x
                break                   # Pula o else.
            x -= 1
        else:
            print y, 'is prime'

prime(13); prime(13.0)
prime 15; prime(15.0)
```

```
prime(3); prime(2)
prime(1); prime(-3)
```

Aqui está o módulo em ação; o operador // permite que ele funcione também para números de ponto flutuante, mesmo que talvez não devesse:

```
% python primes.py
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
3 is prime
2 is prime
1 not prime
-3 not prime
```

Essa função ainda não é muito reutilizável – ela poderia retornar valores, em vez de imprimir –, mas é suficiente para fazer experiências. Ela também ainda não é uma função de números primos matematicamente rigorosa (números de ponto flutuante funcionam) e ainda é ineficiente. Os aprimoramentos são deixados como exercícios para os leitores mais preocupados com a matemática. Dica: um loop `for` sobre `range(y, 1, -1)` pode ser um pouco mais rápido do que o `while` (na verdade, é aproximadamente duas vezes mais rápido na versão 2.2), mas o algoritmo é o gargalo real aqui. Para cronometrar as alternativas, use o módulo interno `time` e padrões de desenvolvimento como aqueles usados neste cronômetro de chamada de função geral (consulte o manual da biblioteca para conhecer os detalhes):

```
def timer(reps, func, *args):
    import time
    start = time.clock()
    for i in xrange(reps):
        apply(func, args)
    return time.clock() - start
```

9. *Abragências de lista.* Aqui está o tipo de código que você deve escrever; podemos ter uma preferência, mas não estamos revelando:

```
>>> values = [2, 4, 9, 16, 25]
>>> import math

>>> res = []
>>> for x in values: res.append(math.sqrt(x))
...
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> map(math.sqrt, values)
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> [math.sqrt(x) for x in values]
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

PARTE V, MÓDULOS

1. *Fundamentos, importação.* Este é mais simples do que você possa imaginar. Quando terminar, seu arquivo e sua interação devem ser muito parecidos com o código a seguir (arquivo *mymod.py*); lembre-se de que o Python pode ler um arquivo inteiro em

uma string ou em uma lista de linhas e que a função interna `len` retorna o comprimento de strings e listas:

```
def countLines(name):
    file = open(name, 'r')
    return len(file.readlines())

def countChars(name):
    return(open(name, 'r').read())

def test(name):                                  # Ou passa objeto arquivo
    return countLines(name), countChars(name)    # Ou retorna um dicionário

% python
>>> import mymod
>>> mymod.test('mymod.py')
(10, 291)
```

No Unix, você pode verificar sua saída com o comando wc; no Windows, dê um clique com o botão direito do mouse em seu arquivo para ver suas propriedades. Mas note que seu script pode relatar menos caracteres do que o Windows relata – por motivos de portabilidade, o Python converte os marcadores de fim de linha `\r\n` do Windows em `\n`, eliminando com isso um byte (caractere) por linha. Para corresponder exatamente às contagens de bytes do Windows, você precisa abrir no modo binário `(rb)` ou adicionar o número de linhas.

A propósito, para fazer a parte "ambiciosa" (passar um objeto arquivo, para abrir o arquivo apenas uma vez), provavelmente você precisará usar o método `seek` do objeto arquivo interno. Não abordamos isso no texto, mas funciona exatamente como a chamada de `fseek` da linguagem C (e a chamada é feita nos bastidores): `seek` reconfigura a posição corrente no arquivo com um deslocamento passado. Após uma chamada de `seek`, as futuras operações de entrada/saída serão relativas à nova posição. Para voltar ao início de um arquivo sem fechar nem abrir novamente, chame `file.seek(0)`; todos os métodos de arquivo `read` começam a atuar a partir da posição corrente do arquivo; portanto, você precisa retroceder para ler novamente. Aqui está como fica essa otimização:

```
def countLines(files):
    file.seek(0)                      # Retrocede para o início do arquivo.
    return len(file.readlines())

def countChars(file):
    file.seek(0)                      # O mesmo (retrocede, se necessário)
    return len(file.read())

def test(name):
    file = open(name, 'r')                        # Passa o objeto arquivo
    return countLines(file), countChars(file)     # Abre o arquivo apenas uma vez.

>>> import mymod2
>>> mymod2.test("mymod2.py")
(11, 392)
```

2. *from / from**. Aqui está a parte `from*`. Substitui `*` por `countChars` para fazer o resto.

```
% python
>>> from mymod import *
```

PARTE VI, CLASSES E POO

1. *Herança.* Aqui está o código da solução desse exercício (arquivo *adder.py*), junto com alguns testes interativos. A sobrecarga de `__add__` precisa aparecer apenas uma vez, na superclasse, pois ela ativa métodos `add` específicos do tipo nas subclasses.

```
class Adder:
    def add(self, x, y):
        print 'not implemented!'
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):                  # Ou nas subclasses?
        return self.add(self.data, other)    # Ou retorno de tipo?

class ListAdder(Adder):
    def add(self, x, y):
        return x + y

class DictAdder(Adder):
    def add(self, x, y):
        new = {}
        for k in x.keys(): new[k] = x[k]
        for k in y.keys(): new[k] = y[k]
        return new
```

```
% python
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
not implemented!
>>> x = ListAdder()
>>> x.add([1], [2])
[1, 2]
>>> x = DictAdder()
>>> x.add({1:1}, {2:2})
{1: 1, 2: 2}

>>> x = Adder([1])
>>> x + [2]
not implemented!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: __add__ nor __radd__ defined for these operands
```

Observe, no último teste, que você recebe um erro para expressões onde uma instância de classe aparece à direita de um operador +; se quiser corrigir isso, use métodos `__radd__`, conforme descrito na seção "Sobrecarga de operador", no Capítulo 21.

Se você estiver salvando um valor na instância de qualquer maneira, também poderia reescrever o método `add` para receber apenas um argumento, no espírito de outros exemplos da Parte IV:

```
% python mysub.py
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 'toast']
['f', 'o', 'o', 'bar']
(3, 2)
```

4. *Métodos de metaclasse.* Resolvemos este exercício como segue. Note que os operadores também tentam buscar atributos por meio de `__getattr__`; você precisa retornar um valor para fazê-los funcionar.

```
>>> class Meta:
...     def __getattr__(self, name):
...         print 'get', name
...     def __setattr__(self, name, value):
...         print 'set', name, value
...
>>> x = Meta()
>>> x.append
get append
>>> x.spam = "pork"
set spam pork
>>>
>>> x + 2
get __coerce__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
>>>
>>> x[1]
get __getitem__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function

>>> x[1:5]
get __len__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
```

5. *Objetos de configuração.* Aqui está o tipo de interação que você deve obter. Os comentários explicam quais métodos são chamados.

```
% python
>>> from setwrapper import Set
>>> x = Set([1,2,3,4])          # Executa __init__
>>> y = Set([3,4,5])

>>> x & y                       # __and__, intersect e, então, __repr__
Set:[3, 4]
>>> x | y                       # __or__, union e, então, __repr__
Set:[1, 2, 3, 4, 5]

>>> z = Set("hello")            # __init__ remove duplicatas.
```

```
>>> z[0], z[-1]                 # __getitem__
('h', 'o')

>>> for c in z: print c,        # __getitem__
...
h e l o
>>> len(z), z                   # __len__, __repr__
(4, Set:['h', 'e', 'l', 'o'])
>>> z & "mello", z | "mello"
(Set:['e', 'l', 'o'], Set:['h', 'e', 'l', 'o', 'm']
```

Nossa solução para a subclasse de extensão de vários operandos é semelhante à classe a seguir (arquivo *multiset.py*). Ela só precisa substituir dois métodos no conjunto original. A string de documentação da classe explica como ela funciona.

```
from setwrapper import Set

class MultiSet(Set)
"""
herda todos os nomes de Set, mas estende intersect
e union para suportar vários operandos; note que "self"
ainda é o primeiro argumento (agora armazenado no
argumento *args); note também que os operadores herdados
& e | chamam os novos métodos aqui com 2 argumentos, mas
processar mais do que 2 exige uma chamada de método
e não uma expressão:
"""

def intersect(self, *others):
    res = []
    for x in self:                  # Percorre a primeira seqüência
        for other in others:        # para todos os outros args.
            if x not in other: break  # Item em cada um?
        else:                       # Não: sai do loop
            res.append(x)           # Sim: adiciona item no final
    return Set(res)

def union(*args):                   # self é args[0].
    res = []
    for seq in args:                # Para todos os args
        for x in seq:               # Para todos os nós
            if not x in res:
                res.append(x)       # Adiciona novos itens no resultado.
    return Set(res)
```

Sua interação com a extensão será algo de acordo com o que segue. Note que você pode interceptar usando `&` ou chamando `intersect`, mas deve chamar `intersect` para três ou mais operandos; `&` é um operador binário (de dois lados). Note também que poderíamos ter chamado `MultiSet` simplesmente de `Set`, para tornar essa alteração mais transparente, se usássemos `setwrapper.Set` para nos referirmos ao original dentro de `multiset`:

```
>>> from multiset import *
>>> x = MultiSet([1,2,3,4])
>>> y = MultiSet([3,4,5])
>>> z = MultiSet([0,1,2])
```

```
    def order(self, foodName):  # Inicia uma simulação de pedido de cliente.
        self.cust.placeOrder(foodName, self.empl)
    def result(self):           # Pergunta ao cliente sobre sua comida
        self.cust.printFood()

class Customer:
    def __init__(self):                         # Inicializa food como None.
        self.food = None
    def placeOrder(self, foodName, employee):   # Faz o pedido com Employee.
        self.food = employee.takeOrder(foodName)
    def printFood(self):                        # Imprime o nome de my food.
        print self.food.name

class Employee:
    def takeOrder(self, foodName):  # Retorna Food, com o nome solicitado.
        return Food(foodName)

class Food:
    def __init__(self, name):       # Armazena o nome da food.
        self.name = name

if __name__ == '__main__':
    x = Lunch()                     # Código de auto-teste
    x.order('burritos')             # Se é executado não é importado
    x.result()
    x.order('pizza')
    x.result()

% python lunch.py
burritos
pizza
```

8. *Hierarquia de animais do zoológico*. Aqui está como desenvolvemos a taxonomia em Python (arquivo *zoo.py*); isso é artificial, mas o padrão de desenvolvimento geral se aplica a muitas estruturas reais – desde GUIs até bancos de dados de funcionários. Note que a referência `self.speak` em Animal provoca uma pesquisa de herança independente, a qual localiza `speak` em uma subclasse. Teste isso interativamente, mediante a descrição do exercício. Tente estender essa hierarquia com novas classes e fazer instâncias de várias classes da árvore.

```
class Animal:
    def reply(self): self.speak()       # De volta para a subclasse
    def speak(self): print 'spam'()     # Mensagem personalizada

class Mammal(Animal):
    def speak(self): print 'huh?'

class Cat(Mammal):
    def speak(self): print 'meow'

class Dog(Mammal):
    def speak(self): print 'bark'

class Primate(Mammal):
    def speak(self): print 'Hello world!'

class Hacker(Primate): pass             # Herda de Primate.
```

9. *O esboço do papagaio morto.* Aqui está como implementamos isso (arquivo *parrot.py*). Observe como o método `line` na superclasse `Actor` funciona: acessando atributos de `self` duas vezes, ele envia o Python de volta para a instância duas vezes e, assim, provoca *duas* pesquisas de herança – `self.name` e `self.says()` localizam informações nas subclasses específicas.

```
class Actor:
    def line(self): print self.name + ':', `self.says()`

class Customer(Actor):
    name = 'customer'
    def says(self): return "that's one ex-bird!"

class Clerk(Actor):
    name = 'clerk'
    def says(self): return "no it isn't..."

class Parrot(Actor):
    name = 'parrot'
    def says(self): return None

class Scene:
    def __init__(self):
        self.clerk = Clerk()              # Incorpora algumas instâncias.
        self.customer = Customer()        # Scene é composta.
        self.subject = Parrot()

    def action(self):
        self.customer.line()              # Delega para incorporado.
        self.clerk.line()
        self.subject.line()
```

PARTE VII, EXCEÇÕES E FERRAMENTAS

1. *try/except.* Nossa versão da função `oops` (arquivo *oops.py*) é a seguinte. Quanto às questões que não precisam de codificação, alterar `oops` para lançar `KeyError` em vez de `IndexError` significa que a exceção não será capturada pela rotina de tratamento de `try` (ela "infiltra" no nível superior e gera a mensagem de erro padrão do Python). Os nomes `KeyError` e `IndexError` vêm do escopo de nomes interno mais externo. Importe `__builtin__` e passe como argumento para a função `dir`, para ver você mesmo.

```
def oops():
    raise IndexError

def doomed():
    try:
        oops()
    except IndexError:
        print 'caught an index error!'
    else:
        print 'no error caught…'

if __name__ == '__main__': doomed()

% python oops.py
caught an index error!
```

```
safe(oops.oops)

% python safe2.py
Traceback (innermost last):
  File "safe2.py", line 5, in safe
    apply(entry, args)           # captura tudo mais
  File "oops.py", line 4, in oops
    raise MyError, 'world'
hello: world
Got hello world
```

PARTE VIII, AS CAMADAS EXTERNAS

Capítulo 27, Tarefas comuns no Python

1. *Evitando expressões regulares.* Este programa é longo e maçante, mas não é particularmente complicado. Veja se você consegue entender como ele funciona. Se isso é mais fácil para você do que as expressões regulares ou não, depende de muitos fatores, como sua familiaridade com as expressões regulares e se você se sente à vontade com as funções do módulo `string`. Use o tipo de programação que funcionar para seu caso.

```
file = open('pepper.txt')
text = file.read()
paragraphs = text.split('\n\n')

def find_indices_for(big, small):
    indices = []
    cum = 0
    while 1:
        index = big.find(small)
        if index == -1:
            return indices
        indices.append(index+cum)
        big = big[index+len(small):]
        cum = cum + index + len(small)

def fix_paragraphs_with_word(paragraphs, word):
    lenword = len(word)
    for par_no in range(len(paragraphs)):
        p = paragraphs[par_no]
        wordpositions = find_indices_for(p, word)
        if wordposition ==[]: return
        for start in wordpositions:
            # Procura 'pepper' antecipadamente.
            indexpepper = p.find('pepper')
            if indexpepper == -1: return -1
            if p[start:indexpepper].strip():
                # Algo diferente de um espaço em branco no meio!
                continue
            where = indexpepper+len('pepper')
            if p[where:where+len('corn')] == 'corn':
                # Isso é seguido imediatamente por 'corn'!
                continue
            if p.find('salad') < where:
                # Isso não é seguido por 'salad'.
```

pode encadear as operações, de modo que, para descobrir qual é a última palavra da terceira linha do terceiro parágrafo, basta escrever:

```
>>> print bigtext.paragraph(2).line(2).word(-1)
'cook'
```

3. *Descrevendo um diretório.* Naturalmente, existem várias soluções para esse exercício. Uma solução simples é:

```
import os, sys, stat

def describedir(start):
    def describedir_helper(arg, dirname, files):
        """ Função auxiliar para descrever diretórios """
        print "Directory %s has files:" % dirname
        for file in files:
            # Localiza o caminho completo até o arquivo (diretório + nome de arquivo).
            fullname = os.path.join(dirname, file)
            if os.path.isdir(fullname):
                # Se for um diretório, indica isso; não precisa descobrir o tamanho.
                print ' '+ file + ' (subdir)'
            else:
                # Descobre o tamanho e imprime a informação.
                size = os.stat(fullname)[stat.ST_SIZE]
                print ' '+file+' size=' + `size`

    # Inicia a 'varredura'.
    os.path.walk(start, describedir_helper, None)
```

que usa a função `walk` no módulo `os.path` e funciona muito bem:

```
>>> import describedir
>>> describedir.describedir2('testdir')
Directory testdir has files:
describedir.py size=939
subdir1 (subdir)
subdir2 (subdir)
Directory test\subdir1 has files:
makezeros.py size=125
subdir3 (subdir)
Directory testdir\subdir1\subdir3 has files:
Directory testdir\subdir2 has files:
```

Note que você poderia ter descoberto o tamanho dos arquivos com `len(open(fullname, 'rb').read())`, mas isso só funciona quando se tem acesso de leitura a todos os arquivos e é muito ineficiente. A chamada de `stat` no módulo `os` fornece todos os tipos de informações úteis em uma tupla e o módulo `stat` define alguns nomes que tornam desnecessário lembrar a ordem dos elementos nessa tupla. Consulte a *Library Reference* para ver os detalhes.

4. *Modificando o prompt.* O segredo deste exercício é lembrar que os atributos `ps1` e `ps2` do módulo `sys` podem ser qualquer coisa, incluindo uma instância de classe com um método `__repr__` ou `__str__`. Por exemplo:

```
import sys, os
class MyPrompt:
```

```
del sys.argv[-1]                          # Tratamos desse argumento.
...                                       # Continua como antes.
```

Capítulo 28, Modelos

1. *Falsificando a Web.* O que você precisa fazer é criar instâncias de uma classe que tenha o atributo `fieldnames` e variáveis de instância apropriadas. Uma possível solução é:

```
class FormData:
    def __init__(self, dict):
        for k, v in dict.items():
            setattr(self, k, v)
class FeedbackData(FormData):
    """ Um FormData gerado pelo formulário comment.html."""
    fieldnames = ('name', 'address', 'email', 'type', 'text')
    def __repr__(self):
        return "%(type)s from %(name)s on %(time)s" % vars(self)

fake_entries = [
    {'name': "John Doe",
    'address': '500 Main St., SF CA 94133',
    'email': 'john@sf.org',
    'type': 'comment',
    'text': 'Great toothpaste!'},
    {'name': "Suzy Doe",
    'address': '500 Main St., SF CA 94133',
    'email': 'suzy@sf.org',
    'type': 'complaint',
    'text': 'It doesn't taste good when I kiss John!'},
    ]

DIRECTORY = r'C:\complaintdir'
if __name__ == '__main__':
    import tempfile, pickle, time
    tempfile.tempdir = DIRECTORY
    for fake_entry in fake_entries:
        data = FeedbackData(fake_entry)
        filename = tempfile.mktemp()
        data.time = time.asctime(time.localtime(time.time()))
        pickle.dump(data, open(filename, 'w'))
```

Conforme você pode ver, a única coisa que realmente teve de mudar foi o funcionamento do construtor de `FormData`, pois ele precisa fazer a configuração dos atributos de um diretório e não de um objeto `FieldStorage`.

2. *Limpeza.* Há muitas maneiras de lidar com esse problema. Uma fácil é modificar o programa *formletter.py* para manter uma lista dos nomes de arquivo que ele já processou (em um arquivo colocado em conserva, é claro!). Isso pode ser feito modificando-se o teste `if __main__ == '__name__'` para algo como o seguinte (as linhas novas estão em negrito):

```
if __name__ == '__main__':
    import os, pickle
    CACHEFILE = 'C:\cache.pik'
    from feedback import DIRECTORY#, FormData, FeedbackData
    if os.path.exists(CACHEFILE):
```

```
        processed_files = pickle.load(open(CACHEFILE))
    else:
        processed_files = []
    for filename in os.listdir(DIRECTORY):
        if filename in processed_files: continue  # Pula esse nome de arquivo.
        Processed_files.append(filename)
        data = pickle.load(open(os.path.join(DIRECTORY, filename)))
        if data.type == 'complaint':
            print "Printing letter for %(name)s." % vars(data)
            print_formletter(data)
        else:
            print "Got comment from %(name)s, skipping printing." %\
                   vars(data)
    pickle.dump(processed_file, open(CACHEFILE, 'w')
```

Conforme pode ver, você simplesmente carrega uma lista dos nomes de arquivo anteriores, se ela existir (e, caso contrário, usa uma lista vazia), e compara os nomes de arquivo com as entradas da lista para determinar o que vai pular. Se você não pular um nome, ele precisa ser adicionado na lista. Finalmente, na saída do programa, coloca a nova lista em conserva.

3. *Adicionando representação gráfica paramétrica em grapher.py.* Este exercício é muito simples, pois basta alterar o código de desenho na classe `Chart`. Especificamente, o código entre `xmin, xmax = 0, N-1` e `graphics.fillPolygon(...)` deve ser colocado em um teste `if`, para que o novo código seja:

```
if not hasattr(self.data[0], '__len__'): # Provavelmente é um número (1D).
       xmin, xmax = 0, N-1
# Código de programa existente, até graphics.fillPolygon(xs, ys, len(xs))
elif len(self.data[0]) == 2:             # trataremos apenas com 2-D
       xmin = reduce(min, map(lambda d: d[0], self.data))
       xmax = reduce(max, map(lambda d: d[0], self.data))

       ymin = reduce(min, map(lambda d: d[1], self.data))
       ymax = reduce(max, map(lambda d: d[1], self.data))

       zero_y = y_offset - int(-ymin/(ymax-ymin)*height)
       zero_x = x_offset + int(-xmin/(xmax-xmin)*width)

       for i in range(N):
           xs[i] = x_offset + int((self.data[i][0]-xmin)/(xmax-xmin)*width)
           ys[i] = y_offset - int((self.data[i][1]-ymin)/(ymax-ymin)*height)
       graphics.color = self.color
       if self.style == "Line":
           graphics.drawPolyline(xs, ys, len(xs))
       else:
           xs.append(xs[0]; ys.append(ys[0])
           graphics.fillPolygon(xs, ys, len(xs))
```

Índice

Símbolo

A

D

J

K

L

M

N

O

P

Q

R

Colofão

Nosso projeto gráfico é o resultado de comentários dos leitores, de nossa própria experiência e do retorno dos canais de distribuição. Capas diferenciadas complementam uma típica abordagem de assuntos técnicos, inspirando personalidade e vida em temas potencialmente áridos.

O animal que aparece na capa deste livro é um rato trocador (*Neotoma*, família dos *Muridae*). Esse rato vive em diferentes habitats (principalmente rochosos, no cerrado e em áreas desérticas), em grande parte das Américas do Norte e Central, geralmente a alguma distância dos seres humanos, embora ocasionalmente cause danos às plantações. Ele é um bom alpinista, faz ninhos em árvores ou arbustos de até 6 m de altura; algumas espécies fazem covas subterrâneas, em rachaduras nas pedras ou vivem em buracos abandonados por outras espécies.

Esses roedores bege-acinzentados, de tamanho médio, são os ratos trocadores originais: eles carregam e escondem pequenos objetos em suas casas, sejam necessários ou não, e têm uma atração especial por objetos brilhantes, como latas de conserva, vidro e prataria.

Matt Hutchinson foi o editor executivo deste livro*. A Argosy Publishing fez a produção. Collen Gorman, Emily Quill e Mary Anne Mayo ficaram com o controle de qualidade.

Edie Freedman concebeu a capa deste livro. A imagem da capa é uma gravura do século 19 da *Cuvier's Animals*. Emma Colby produziu o layout da capa com QuarkXPress 4.1, usando a fonte ITC Garamond da Adobe.

David Futato fez o projeto gráfico interno da edição americana. Este livro foi convertido para FrameMaker 5.5.6 por Julie Hawks, com uma ferramenta de conversão de formato criada por Erik Ray, Jason McIntosh, Neil Walls e Mike Sierra, que usa tecnologias Perl e XML. A fonte do texto é Linotype Birka, a fonte dos cabeçalhos é Adobe Myriad Condensed e a fonte do código é TheSans Mono Condensed da LucasFont. As ilustrações que aparecem no livro foram produzidas por Robert Romano e Jessamyn Read, usando Macromedia FreeHand 9 e Adobe Photoshop 6. Os ícones de dica e alerta foram desenhados por Christopher Bing. Este colofão foi escrito por Nancy Kotary.

*N. de R. As informações aqui constantes dizem respeito à edição americana da obra.

Bookman® Editora
Av. Jerônimo de Ornelas, 670
90040-340 Porto Alegre, RS, Brasil
Fone (51) 3027-7000 Fax (51) 3027-7070
e-mail: bookman@artmed.com.br

O'REILLY®

Recorte e use como um marcador de página

LINGUAGEM DE PROGRAMAÇÃO

ARNOLD, GOSLING & HOLMES
A Linguagem de Programação Java, 4.ed.

DEITEL & DEITEL
C++, Como Programar, 3.ed.

DEITEL, DEITEL, NIETO, LIN & SADHU
XML, Como Programar

DEITEL, DEITEL, NIETO & MCPHIE
Perl, Como Programar

FLANAGAN, D.
Java: O Guia Essencial, 5.ed.

FLANAGAN, D.
JavaScript: O Guia Definitivo, 4.ed

GALUPPO, SANTOS & MATHEUS
Desenvolvendo com C#

*HERRINGTON, J. D.
Dicas de PHP

HORSTMANN, C.
Big Java

HORSTMANN, C.
Conceitos de Computação com o Essencial de C++, 3. ed

HORSTMANN, C.
Conceitos de Computação com o Essencial de Java, 3. ed

HUBBARD, J.
Programação em C++, 2.ed.
(COLEÇÃO SCHAUM)

HUBBARD, J.
Programação com Java, 2.ed.
(COLEÇÃO SCHAUM)

LIPPMAN, S.
C#: Um Guia Prático

LUTZ & ASCHER
Aprendendo Python, 2.ed.

*ROBBINS & BEEBE
Classic Shell Scripting

ROMAN, AMBLER & JEWELL
Dominando o Enterprise JavaBeans, 2.ed.

SIEVER & COLS
Linux: O Guia Essencial

STROUSTRUP, B.
A Linguagem de Programação C++, 3.ed.

TITTEL, E.
XML (COLEÇÃO SCHAUM)

* *Livros em produção no momento da impressão desta obra, mas que muito em breve estarão à disposição dos leitores em língua portuguesa.*

O'REILLY®

Aprendendo Python

Python é uma linguagem de programação popular orientada a objetos e de código-fonte aberto, usada para programas independentes e para aplicações de script. É portável, poderosa e muito fácil de usar. E não há maneira mais rápida de dominá-la do que aprender com especialistas.

Esta nova edição de *Aprendendo Python* coloca você nas mãos de Mark Lutz e David Ascher, dois conhecidos instrutores de Python que, com um texto amigável e bem estruturado, levaram muitos programadores à proficiência na linguagem.

Completamente atualizado, o livro apresenta os elementos básicos da versão mais recente do Python, o 2.3, abordando novos recursos, como abrangências de lista, escopos aninhados e iteradores/funções geradoras.

Além dos recursos da linguagem, esta edição também inclui contexto novo para programadores menos experientes, com novos panoramas da programação orientada a objetos e tipagem dinâmica, novas discussões sobre execução de programas e opções de configuração, nova cobertura de fontes de documentação e muito mais. Casos de uso completos tornam a aplicação dos recursos da linguagem mais concretos.

ISBN 978-85-7780-013-1